Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging

Niv Dayan, Stratos Idreos
Harvard University

ABSTRACT

We show that all mainstream LSM-tree based key-value stores in the literature and in industry suboptimally trade between the I/O cost of updates on one hand and the I/O cost of lookups and storage space on the other. The reason is that they perform equally expensive merge operations across all levels of LSM-tree to bound the number of runs that a lookup has to probe and to remove obsolete entries to reclaim storage space. With state-of-the-art designs, however, merge operations from all levels of LSM-tree but the largest (i.e., most merge operations) reduce point lookup cost, long range lookup cost, and storage space by a negligible amount while significantly adding to the amortized cost of updates.

To address this problem, we introduce Lazy Leveling, a new design that removes merge operations from all levels of LSM-tree but the largest. Lazy Leveling improves the worst-case complexity of update cost while maintaining the same bounds on point lookup cost, long range lookup cost, and storage space. We further introduce Fluid LSM-tree, a generalization of the entire LSM-tree design space that can be parameterized to assume any existing design. Relative to Lazy Leveling, Fluid LSM-tree can optimize more for updates by merging less at the largest level, or it can optimize more for short range lookups by merging more at all other levels.

We put everything together to design Dostoevsky, a key-value store that adaptively removes superfluous merging by navigating the Fluid LSM-tree design space based on the application workload and hardware. We implemented Dostoevsky on top of RocksDB, and we show that it strictly dominates state-of-the-art designs in terms of performance and storage space.

ACM Reference Format:
Niv Dayan, Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging . In *Proceedings of 2018 International Conference on Management of Data (SIGMOD'18)*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196927

1 INTRODUCTION

Key-Value Stores and LSM-Trees. A key-value store is a database that efficiently maps from search keys to their corresponding data values. Key-value stores are used everywhere today from graph processing in social media [8, 17] to event log processing in cyber security [18] to online transaction processing [27]. To persist key-value entries in storage, most key-value stores today use LSM-tree [41]. LSM-tree buffers inserted/updated entries in main memory and flushes the buffer as a sorted run to secondary storage every time that it fills up. LSM-tree later sort-merges these runs to bound the number of runs that a lookup has to probe and to remove obsolete entries, i.e., for which there exists a more recent entry with the same key. LSM-tree organizes runs into levels of exponentially increasing capacities whereby larger levels contain older runs. As entries are updated out-of-place, a point lookup finds the most recent version of an entry by probing the levels from smallest to largest and terminating when it finds the target key. A range lookup, on the other hand, has to access the relevant key range from across all runs at all levels and to eliminate obsolete entries from the result set. To speed up lookups on individual runs, modern designs maintain two additional structures in main memory. First, for every run there is a set of fence pointers that contain the first key of every block of the run; this allows lookups to access a particular key within a run with just one I/O. Second, for every run there exists a Bloom filter; this allows point lookups to skip runs that do not contain the target key. This overall design is adopted in a large number of modern key-value stores including LevelDB [32] and BigTable [19] at Google, RocksDB [28] at Facebook, Cassandra [34], HBase [7] and Accumulo [5] at Apache, Voldemort [38] at LinkedIn, Dynamo [26] at Amazon, WiredTiger [52] at MongoDB, and bLSM [48] and cLSM [31] at Yahoo. Relational databases today such as MySQL (using MyRocks [29]) and SQLite4 support this design too as a storage engine by mapping primary keys to rows as values.

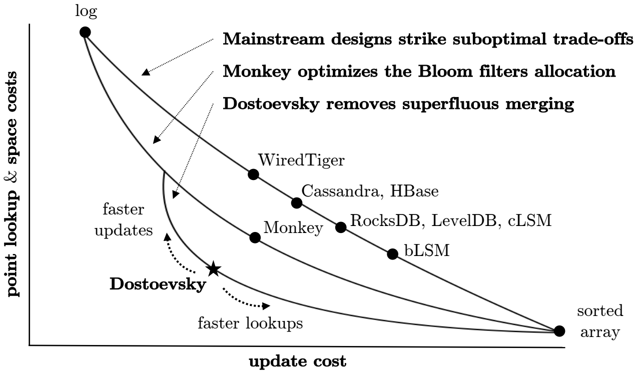


Figure 1: Dostoevsky enables richer space-time trade-offs among updates, point lookups, range lookups and space-amplification, and it navigates the design space to find the best trade-off for a particular application.

The Problem. The frequency of merge operations in LSM-tree controls an intrinsic trade-off between the I/O cost of updates on one hand and the I/O cost of lookups and storage space-amplification

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196927

(i.e., caused by the presence of obsolete entries) on the other. The problem is that existing designs trade suboptimally among these metrics. Figure 1 conceptually depicts this by plotting point lookup cost and space-amplification on the y-axis against update cost on the x-axis (while these y-axis metrics have different units, their trade-off curves with respect to the x-axis have the same shape). The two points at the edges of the curves are a log and a sorted array. LSM-tree degenerates into these edge points when it does not merge at all or when it merges as much as possible, respectively. We place mainstream systems along the top curve between these edge points based on their default merge frequencies, and we draw a superior trade-off curve for Monkey [22], which represents the current state of the art. We show that there exists an even superior trade-off curve to Monkey. Existing designs forgo a significant amount of performance and/or storage space for not being designed along this bottom curve.

The Problem's Source. By analyzing the design space of state-of-the-art LSM-trees, we pinpoint the problem to the fact that the worst-case update cost, point lookup cost, range lookup cost, and space-amplification derive differently from across different levels.

- **Updates.** The I/O cost of an update is paid later through the merge operations that the updated entry participates in. While merge operations at larger levels entail exponentially more work, they take place exponentially less frequently. Therefore, updates derive their I/O cost equally from merge operations across all levels.
- **Point lookups.** While mainstream designs along the top curve in Figure 1 set the same false positive rate to Bloom filters across all levels of LSM-tree, Monkey, the current state of the art, sets exponentially lower false positive rates to Bloom filters at smaller levels [22]. This is shown to minimize the sum of false positive rates across all filters and to thereby minimize I/O for point lookups. At the same time, this means that access to smaller levels is exponentially less probable. Therefore, most point lookup I/Os target the largest level.
- **Long range lookups**[1]**.** As levels in LSM-tree have exponentially increasing capacities, the largest level contains most of the data, and so it tends to contain most of the entries within a given key-range. Therefore, most I/Os issued by long range lookups target the largest level.
- **Short range lookups.** Range lookups with extremely small key ranges only access approximately one block within each run regardless of the run's size. As the maximum number of runs per level is fixed in state-of-the-art designs, short range lookups derive their I/O cost equally from across all levels.
- **Space-Amplification.** The worst-case space-amplification occurs when all entries at smaller levels are updates to entries at the largest level. Therefore, the highest fraction of obsolete entries in the worst-case is at the largest level.

Since the worst-case point lookup cost, long range lookup cost and space-amplification derive mostly from the largest level, merge operations at all levels of LSM-tree but the largest (i.e., most merge operations) hardly improve on these metrics while significantly adding to the amortized cost of updates. This leads to suboptimal trade-offs. We solve this problem from the ground up in three steps.

Solution 1: Lazy Leveling to Remove Superfluous Merging. We expand the LSM-tree design space with Lazy Leveling, a new design that removes merging from all but the largest level of LSM-tree. Lazy Leveling improves the worst-case cost complexity of updates while maintaining the same bounds on point lookup cost, long range lookup cost, and space-amplification and while providing a competitive bound on short range lookup cost. We show that the improved update cost can be traded to reduce point lookup cost and space-amplification. This generates the bottom curve in Figure 1, which offers richer space-time trade-offs that have been impossible to achieve with state-of-the-art designs until now.

Solution 2: Fluid LSM-Tree for Design Space Fluidity. We introduce Fluid LSM-tree as a generalization of LSM-tree that enables transitioning fluidly across the whole LSM-tree design space. Fluid LSM-tree does this by controlling the frequency of merge operations separately for the largest level and for all other levels. Relative to Lazy Leveling, Fluid LSM-tree can optimize more for updates by merging less at the largest level, or it can optimize more for short range lookups by merging more at all other levels.

Solution 3: Dostoevsky to Navigate the Design Space. We put everything together in **Do**stoevsky: **S**pace-**T**ime **O**ptimized **Ev**olvable **S**calable **Key**-Value Store. Dostoevsky analytically finds the tuning of Fluid LSM-tree that maximizes throughput for a particular application workload and hardware subject to a user constraint on space-amplification. It does this by pruning the search space to quickly find the best tuning and physically adapting to it during runtime. Since Dostoevsky spans all existing designs and is able to navigate to the best one for a given application, it strictly dominates existing key-value stores in terms of performance and space-amplification. We depict Dostoevsky in Figure 1 as a black star that can navigate the entire design space.

Contributions. Our contributions are summarized below.

- We show that state-of-the-art LSM-trees perform equally expensive merge operations across all levels of LSM-tree, yet merge operations at all but the largest level (i.e., most merge operations) improve point lookup cost, long range lookup cost, and space-amplification by a negligible amount while adding significantly to the amortized cost of updates.
- We introduce Lazy Leveling to remove merge operations at all but the largest level. This improves the cost complexity of updates while maintaining the same bounds on point lookups, long range lookups, and space-amplification and while providing a competitive bound on short range lookups.
- We introduce Fluid LSM-tree, a generalization of LSM-tree that spans all existing designs. Relative to Lazy Leveling, Fluid LSM-tree can optimize more for updates by merging less at the largest level, or it can optimize more for short range lookups by merging more at all other levels.
- We introduce Dostoevsky, a key-value store that dynamically adapts across the Fluid LSM-tree design space to the design that maximizes the worst-case throughput based on the application workload and the hardware subject to a constraint on space-amplification.
- We implemented Dostoevsky on RocksDB and show that it dominates existing designs for any application scenario.

[1] In Section 3, we distinguish formally between short and long range lookups.

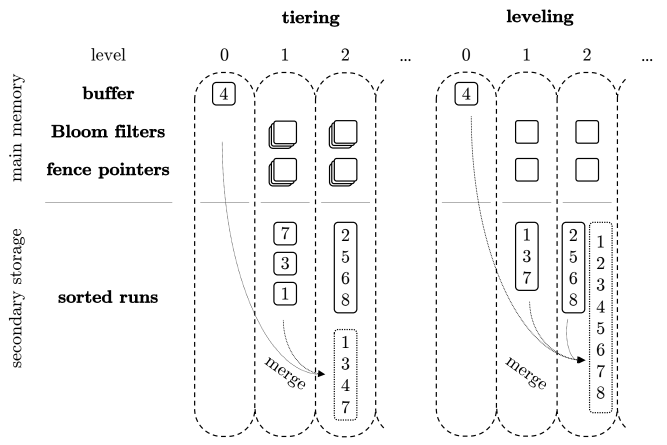


Figure 2: An overview of an LSM-tree using the tiering and leveling merge policies.

Term	Definition	Unit
N	total number of entries	entries
L	number of levels	levels
L_{max}	maximum possible number of levels	levels
B	number of entries that fit into a storage block	entries
P	size of the buffer in storage blocks	blocks
T	size ratio between adjacent levels	
T_{lim}	size ratio at which L converges to 1	
M	main memory allocated to the Bloom filters	bits
p_i	false positive rate for Bloom filters at Level i	%
s	selectivity of a range lookup	%
R	zero-result point lookup cost	I/Os
V	non-zero-result point lookup cost	I/Os
Q	range lookup cost	I/Os
W	update cost	I/Os
K	bound on number of runs at Levels 1 to $L-1$	runs
Z	bound on number of runs at Level L	runs
μ	storage sequential over random access speed	
ϕ	storage write over read speed	

Table 1: Table of terms used throughput the paper.

2 BACKGROUND

This section gives the necessary background on LSM-tree based key-value stores. Figure 2 illustrates their architecture and Table 1 gives a list of terms we use throughout the paper.

LSM-Tree Structure. LSM-tree optimizes for write-heavy workloads. This is an important performance goal for systems today because the proportion of application writes is continuously increasing (e.g., in 2012 Yahoo! reported that the proportion of writes targeting their web-service was 50% and projected to continue accelerating [48]). To optimize for writes, LSM-tree initially buffers all updates, insertions, and deletes (henceforth referred to as updates unless otherwise mentioned) in main memory, as shown in Figure 2. When the buffer fills up, LSM-tree flushes the buffer to secondary storage as a sorted run. LSM-tree sort-merges runs in order to (1) bound the number of runs that a lookup has to access in secondary storage, and to (2) remove obsolete entries to reclaim space. It organizes runs into L conceptual levels of exponentially increasing sizes. Level 0 is the buffer in main memory, and runs belonging to all other levels are in secondary storage.

The balance between the I/O cost of merging and the I/O cost of lookups and space-amplification can be tuned using two knobs. The first knob is the size ratio T between the capacities of adjacent levels; T controls the number of levels of LSM-tree and thus the overall number of times that an entry gets merged *across* levels. The second knob is the merge policy, which controls the number of times an entry gets merged *within* a level. All designs today use either one of two merge policies: tiering or leveling (e.g., Cassandra and RocksDB use tiering and leveling by default, respectively [28, 34]). With tiering, we merge runs within a level only when the level reaches capacity [33]. With leveling, we merge runs within a level whenever a new run comes in [41]. We compare these policies in Figure 2 (with a size ratio of 4 and a buffer size of one entry[2]). In both cases, the merge is triggered by the buffer flushing and causing Level 1 to reach capacity. With tiering, all runs at Level 1 get merged into a new run that gets placed at Level 2. With leveling, the merge also includes the preexisting run at Level 2. We formally study this design space in the next section. We discuss further details of merge mechanics in Appendix D and other log-structured designs in Appendix E.

Number of Levels. The buffer at Level 0 has a capacity of $B \cdot P$ entries, where B is the number of entries that fit into a storage block, and P is the size of the buffer in terms of storage blocks. In general, Level i has a capacity of $B \cdot P \cdot T^i$ entries, and the capacity at the largest level can be approximated as having $N \cdot \frac{T-1}{T}$ entries. To derive the number of levels, we divide the capacity at the largest level by the capacity of the buffer and take log base T of the quotient, as shown in Equation 1.

$$L = \left\lceil \log_T \left(\frac{N}{B \cdot P} \cdot \frac{T-1}{T} \right) \right\rceil \tag{1}$$

We restrict the size ratio to the domain of $2 \le T \le T_{lim}$, where T_{lim} is defined as $\frac{N}{B \cdot P}$. As the size ratio increases and approaches T_{lim}, the number of levels decreases and approaches 1. Increasing T beyond T_{lim} has no structural impact. Furthermore, restricting T to be 2 or greater ensures that the resulting run from a merge operation at level i is never large enough to move beyond level $i+1$. In other words, this ensures that runs do not skip levels. Thus, the highest possible number of levels L_{max} is $\lceil \log_2 \left(\frac{N}{B \cdot P} \cdot \frac{1}{2} \right) \rceil$ (i.e., when the size ratio is set to 2).

Finding Entries. Since entries are updated out-of-place, multiple versions of an entry with the same key may exist across multiple levels (and even across runs within a level with tiering). To ensure that a lookup is always able to find the most recent version of an entry, LSM-tree takes the following measures. (1) When an entry is inserted into the buffer and the buffer already contains an entry with the same key, the newer entry replaces the older one. (2) When two runs that contain an entry with the same key are merged, only the entry from the newer run is kept because it is more recent. (3) To be able to infer the order at which different entries with the same key across different runs were inserted, a run can only be merged with the next older or the next younger run. Overall, these rules

[2] In practice, the buffer size is typically set between 2 and 64 MB [28, 32], but we use a size of one entry in this example for ease of illustration.

ensure that if there are two runs that contain different versions of the same entry, the younger run contains the newer version.

Point Lookups. A point lookup finds the most recent version of an entry by traversing the levels from smallest to largest, and runs within a level from youngest to oldest with tiering. It terminates when it finds the first entry with a matching key.

Range Lookups. A range lookup has to find the most recent versions of all entries within the target key range. It does this by sort-merging the relevant key range across all runs at all levels. While sort-merging, it identifies entries with the same key across different runs and discards older versions.

Deletes. Deletes are supported by adding a one-bit flag to every entry. If a lookup finds that the most recent version of an entry has this flag on, it does not return a value to the application. When a deleted entry is merged with the oldest run, it is discarded as it has replaced all entries with the same key that were inserted prior to it.

Fragmented Merging. To smooth out performance slumps due to long merge operations at larger levels, mainstream designs partition runs into files (e.g., 2 to 64 MB [28, 32]) called Sorted String Tables (SSTables), and they merge one SSTable at a time with SSTables with an overlapping key range at the next older run. This technique does not affect the worst-case I/O overhead of merging but only how this overhead gets scheduled across time. For readability throughout the paper, we discuss merge operations as having the granularity of runs, though they can also have the granularity of SSTables.

Space-Amplification. The factor by which the presence of obsolete entries amplify storage space is known as space-amplification. Space-amplification has traditionally not been a major concern for data structure design due to the affordability of disks. The advent of SSDs, however, makes space-amplification an important cost concern (e.g., Facebook has recently switched from B-trees to leveled LSM-trees due to their superior space-amplification properties [27]). We include space-amplification as a cost metric to give a complete picture of the designs that we introduce and evaluate.

Fence Pointers. All major LSM-tree based key-value stores index the first key of every block of every run in main memory. We call these fence pointers (see Figure 2). Formally, the fence pointers take up $O(N/B)$ space in main memory, and they enable a lookup to find the relevant key-range at every run with one I/O.

Bloom Filters. To speed up point lookups, which are common in practice [16, 48], each run has a Bloom filter [14] in main memory, as shown in Figure 2. A Bloom filter is a space-efficient probabilistic data structure used to answer set membership queries. It cannot return a false negative, though it returns a false positive with a tunable false positive rate (FPR). The FPR depends on the ratio between the number of *bits* allocated to the filter and the number of *entries* in the set according to the following expression[3] [50]:

$$FPR = e^{-(bits/entries)\cdot \ln(2)^2} \qquad (2)$$

A point lookup probes a Bloom filter before accessing the corresponding run in storage. If the filter returns a true positive, the lookup accesses the run with one I/O (i.e., using the fence pointers), finds the matching entry, and terminates. If the filter returns a negative, the lookup skips the run thereby saving one I/O. Otherwise, we have a false positive, meaning the lookup wastes one I/O by accessing the run, not finding a matching entry, and having to continue searching for the target key in the next run.

Bloom filter has a useful property that if it is partitioned into smaller equally-sized Bloom filters with an equal division of entries among them, the FPR of each one of the new partitioned Bloom filters is asymptotically the same as the FPR of the original filter (though slightly higher in practice) [50]. For ease of discussion, we refer to Bloom filters as being non-partitioned, though they can also be partitioned (e.g., per every block of every run) as some designs in industry to enable greater flexibility for space management (e.g., Bloom filters for blocks that are not frequently read by point lookups can be offloaded to storage to save memory) [32].

Applicability Beyond Key-Value Stores. In accordance with designs in industry, our discussion assumes that a key is stored adjacently to its value within a run [28, 32]. For readability, all figures in this paper depict entries as keys, but they represent key-value pairs. Our work also applies to applications where there are no values (i.e., the LSM-tree is used to answer set-membership queries on keys), where the values are pointers to data objects stored outside of LSM-tree [39], or where LSM-tree is used as a building block for solving a more complex algorithmic problem (e.g., graph analytics [17], flash translation layer design [23], etc.). We restrict the scope of analysis to the basic operations and size of LSM-tree so that it can easily be applied to each of these other cases.

3 DESIGN SPACE AND PROBLEM ANALYSIS

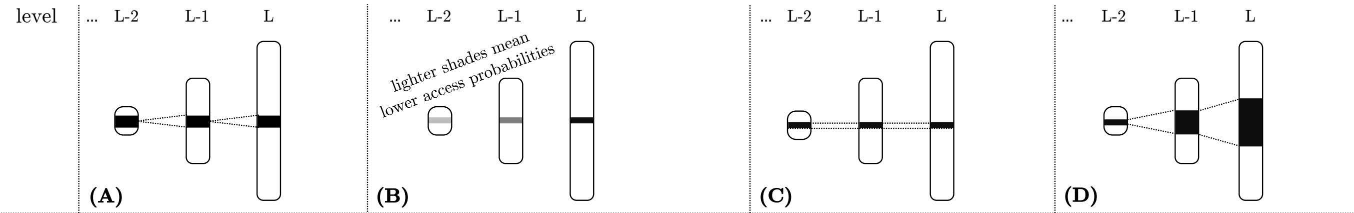
We now analyze how the worse-case space-amplification and I/O costs of updates and lookups derive from across different levels with respect to the merge policy and size ratio. To analyze updates and lookups, we use the disk access model [1] to count the number of I/Os per operation, where an I/O is a block read or written from secondary storage. The results are summarized in Figures 3 and 4.

Analyzing Updates. The I/O cost of updating an entry is paid through the subsequent merge operations that the updated entry participates in. Our analysis assumes a worst-case workload whereby all updates target entries at the largest level. This means that an obsolete entry does not get removed until its corresponding updated entry has reached the largest level. As a result, every entry gets merged across all levels (i.e., rather than getting discarded at some smaller level by a more recent entry and thereby reducing overhead for later merge operations).

With tiering, an entry gets merged $O(1)$ time per level across $O(L)$ levels. We divide this by the block size B since every I/O during a merge operation copies B entries from the original runs to the new run. Thus, the amortized I/O cost for one update is $O(\frac{L}{B})$ I/O.

With leveling, the j^{th} run that arrives at Level i triggers a merge operation involving the existing run at Level i, which is the merged product of the previous $T - j$ runs that arrived since the last time Level i was empty. Overall, an entry gets merged on average $\frac{T}{2}$, or $O(T)$, times per level before that level reaches capacity, and across $O(L)$ levels for a total of $O(T \cdot L)$ merge operations. As with tiering, we divide this by the block size B to get the amortized I/O cost for one update: $O(\frac{L \cdot T}{B})$ I/O.

[3] Equation 2 assumes that the Bloom filter is using the optimal number of hash functions $\frac{bits}{entries} \cdot \ln(2)$ to minimize the false positive rate.



	updates	point lookups	short range lookups	long range lookup
leveling	... $O(\frac{T}{B})$ + $O(\frac{T}{B})$ + $O(\frac{T}{B})$ $=\mathbf{O(\frac{L \cdot T}{B})}$	... $O\left(\frac{e^{-M/N}}{T^2}\right)+O\left(\frac{e^{-M/N}}{T^1}\right)+O\left(\frac{e^{-M/N}}{T^0}\right)=\mathbf{O(e^{-M/N})}$	... $O(1)$ + $O(1)$ + $O(1)$ $= \mathbf{O(L)}$	... $O(\frac{s}{T^2 \cdot B})+O(\frac{s}{T^1 \cdot B})+O(\frac{s}{T^0 \cdot B}) = \mathbf{O(\frac{s}{B})}$
tiering	... $O(\frac{1}{B})$ + $O(\frac{1}{B})$ + $O(\frac{1}{B})$ $=\mathbf{O(\frac{L}{B})}$	... $O\left(\frac{e^{-M/N}}{T^1}\right)+O\left(\frac{e^{-M/N}}{T^0}\right)+O\left(\frac{e^{-M/N}}{T^{-1}}\right)=\mathbf{O(T \cdot e^{-M/N})}$	... $O(T)$ + $O(T)$ + $O(T)$ $= \mathbf{O(L \cdot T)}$	... $O(\frac{s}{T^1 \cdot B})+O(\frac{s}{T^0 \cdot B})+O(\frac{s}{T^{-1} \cdot B})= \mathbf{O(\frac{s \cdot T}{B})}$

Figure 3: In the worst-case, updates and short range lookups derive their I/O cost equally from access to all levels, whereas point lookups and long range lookups derive their I/O cost mostly from access to the largest level.

We now take a closer look at how update cost derives from across different levels. With tiering, Level i fills up every $B \cdot P \cdot T^i$ application updates, and the resulting merge operation copies $B \cdot P \cdot T^i$ entries. With leveling, a merge operation takes place at Level i every $B \cdot P \cdot T^{i-1}$ updates (i.e., every time that a new run comes in), and it copies on average $\frac{B \cdot P \cdot T^i}{2}$ entries. By dividing the number of copied entries by the frequency of a merge operation at Level i for either leveling or tiering, we observe that in the long run the amount of work done by merge operations at every level is the same, as shown with the cost breakdown in Figure 3 (A). The intuition is that while merge operations at larger levels do exponentially more work, they are also exponentially less frequent.

Analyzing Point Lookups. To analyze the worst-case point lookup cost, we focus on zero-result point lookups (i.e., to non-existing keys) because they maximize the long-run average of wasted I/Os (i.e., to runs that do not contain a target key). Zero-result point lookups are common in practice (e.g., in insert-if-not-exist operations) [16, 48], and we extend their analysis to point lookups that target existing entries in the next section.

The I/O cost for a zero-result point lookup is highest when every Bloom filter returns a false positive. In this case, a point lookup issues one I/O to every run, amounting to $O(L)$ wasted I/Os with leveling and $O(T \cdot L)$ wasted I/Os with tiering. In practice, however, the Bloom filters eliminate most I/Os to runs that do not contain the target key; key-value stores in industry use 10 bits per entry for every Bloom filters leading to a false positive rate (FPR) of $\approx$ 1% for each filter [28, 32, 34]. For this reason, we focus on the *expected worst-case* point lookup cost, which estimates the number of I/Os issued by point lookups as a long-run average with respect to the Bloom filters' FPRs. We estimate this cost as the sum of FPRs across all the Bloom filters. The reason is that the I/O cost of probing an individual run is an independent random variable with an expected value equal to the corresponding Bloom filter's FPR, and the expected sum of multiple independent random variables is equal to the sum of their individual expected values [44].

In key-value stores in industry, the number of bits per entry for Bloom filters across all levels is the same [7, 28, 32, 34, 48]. Therefore, the Bloom filter(s) at the largest level are larger than the filters at all smaller levels combined, as they represent exponentially more entries. Thus, the FPR p_L at the largest level is upper bounded using Equation 2 to $O(e^{-M/N})$, where M is the memory budget for all filters and N is the number of entries in the system. Since all other FPRs are the same as p_L, the expected point lookup cost is the product of p_L and the number of runs in the system: $O(e^{-M/N} \cdot L)$ I/Os with leveling and $O(e^{-M/N} \cdot L \cdot T)$ I/Os with tiering.

The most recent paper on this issue named Monkey [22] shows that setting the same number of bits per entry for filters across all levels does not minimize the expected number of wasted I/Os. Instead, Monkey reallocates $\approx$ 1 bit per entry from the filter(s) at the largest level, and it uses these bits to set the number of bits per entry across smaller levels as an increasing arithmetic progression: Level i gets $a + b \cdot (L - i)$ bits per entry, where a and b are small constants. This causes a small, asymptotically constant increase to the FPR at the largest level and an exponential decrease to the FPRs across smaller levels, as they contain exponentially less entries. Since the FPRs are exponentially decreasing for smaller levels, the sum of FPRs converges to a multiplicative constant that is independent of the number of levels. As a result, Monkey shaves a factor of L from the complexity of point lookups leading to $O(e^{-M/N})$ I/Os with leveling and $O(e^{-M/N} \cdot T)$ I/Os with tiering, as we illustrate in Figure 3 (B). It is always beneficial to use Monkey, for zero and non-zero result point lookups alike and with any kind of skew [22].

Overall, we observe that point lookup cost using Monkey derives mostly from the largest level, as smaller levels have exponentially lower FPRs and so access to them is exponentially less probable. As we will see at the end of this section, this opens up avenues for further optimization.

Analyzing Space-Amplification. We define space-amplification as the factor *amp* by which the overall number of entries N is greater than the number of unique entries *unq*: $amp = \frac{N}{unq} - 1$. To analyze the worst-case space-amplification, we observe that levels 1 to $L - 1$ of LSM-tree comprise a fraction of $\frac{1}{T}$ of its capacity whereas level L comprises the remaining fraction of $\frac{T-1}{T}$ of its capacity. With leveling, the worst-case space-amplification occurs when entries at Levels 1 to $L - 1$ are all updates to different entries at Level L thereby rendering at most a fraction of $\frac{1}{T}$ entries at level L obsolete[4]. Space-amplification is therefore $O(\frac{1}{T})$, as shown in Figure 4 (A). For example, in production environments using SSDs at Facebook, RocksDB uses leveling and a size ratio of 10 to bound space-amplification to $\approx$ 10% [27]. With tiering, the worst-case occurs when entries at Levels 1 to $L - 1$ are all updates to different

[4] In fact, if the largest level is not filled to capacity, space-amplification may be higher than $\frac{1}{T}$, but it is straightforward to dynamically enforce the size ratio across levels to T during runtime by adjusting the capacities at levels 1 to $L - 1$ to guarantee the upper bound of $O(\frac{1}{T})$ as in RocksDB [27].

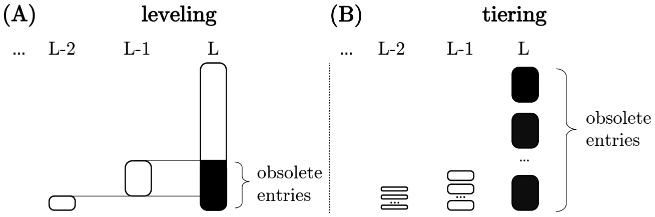


Figure 4: Space-amplification is $O(\frac{1}{T})$ with leveling and $O(T)$ with tiering.

entries at Level L, and where every run at Level L contains the same set of entries. In this case, Level L entirely consists of obsolete entries, and so space-amplification is $O(T)$ as level L is larger by a factor of $T-1$ than all other levels combined. Overall, space-amplification with both leveling and tiering in the worst case derives mostly from the presence of obsolete entries at the largest level.

Analyzing Range Lookups. We denote the selectivity of a range lookup s as the number of unique entries across all runs that fall within the target key range. A range lookup scans and sort-merges the target key range across all runs, and it eliminates obsolete entries from the result set. For analysis, we consider a range lookup to be *long* if the number of blocks accessed is at least twice as large as the maximum possible number of levels: $\frac{s}{B} > 2 \cdot L_{max}$. Under uniformly randomly distributed updates, this condition implies with a high probability that most entries within a target key range are at the largest level. For all practical purposes, we generalize the treatment of long and short range lookups in Section 4.2.

A short range lookup issues approximately one I/O to every run amounting to $O(T \cdot L)$ I/Os with tiering and $O(L)$ I/Os with leveling, as shown in Figure 3 (C). For a long range lookup, the size of the result set before eliminating obsolete entries is on average the product of its selectivity and space-amplification. We divide this product by the block size to get the I/O cost: $O(\frac{T \cdot s}{B})$ with tiering and $O(\frac{s}{B})$ with leveling, as shown in Figure 3 (D).

A key distinction is that a short range lookup derives its cost approximately equally from across all levels, whereas a long range lookup derives most of its cost from access to the largest level.

Mapping the Design Space to the Trade-Off Space. There is an intrinsic trade-off between update cost on one hand and the costs of lookups and space-amplification on the other. We illustrate this trade-off in conceptual Figure 5, whereon the solid line plots the different costs of lookups and space-amplification on the y-axis against update cost on the x-axis for both leveling and tiering as we vary the size ratio, all based on the properties in Figures 3 and 4. When the size ratio is set to its limiting value of T_{lim} (meaning there is only one level in storage), a tiered LSM-tree degenerates into a log whereas a leveled LSM-tree degenerates into a sorted array. When the size ratio is set to its lower limit of 2, the performance characteristics for leveling and tiering converge as their behaviors become identical: the number of levels is the same and a merge operation is triggered at every level when the second run comes in. In general, as the size ratio increases with leveling/tiering, lookup cost and space-amplification decrease/increase and update cost increases/decreases. Thus, the trade-off space is partitioned: leveling has strictly better lookup costs and space-amplification and strictly worse update cost than tiering.

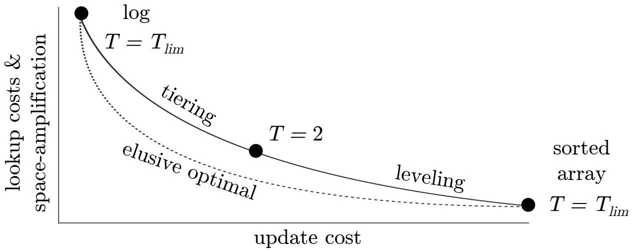


Figure 5: How far down is it possible to push the curve of possible space-time trade-offs?

The Holy Grail. The solid line in Figure 5 reflects the properties of Monkey, the current state of the art. Figure 5 also shows a dotted line labeled the *elusive optimal*. The question guiding our research is whether other designs are possible with space-time trade-offs that more closely approach or even reach the elusive optimal.

The Opportunity: Removing Superfluous Merging. We have identified an asymmetry: point lookup cost, long range lookup cost, and space-amplification derive mostly from the largest level, while update cost derives equally from across all levels. This means that merge operations at smaller levels significantly amplify update cost while yielding a comparatively insignificant benefit for space-amplification, point lookups, and long range lookups. There is therefore an opportunity of a merge policy that merges less at smaller levels.

4 LAZY LEVELING, FLUID LSM-TREE, AND DOSTOEVSKY

We now present Lazy Leveling, Fluid LSM-Tree, and Dostoevsky to fluidly and dynamically adapt across an expanded LSM-tree design space with richer performance and space trade-offs.

4.1 Lazy Leveling

Lazy Leveling is a merge policy that eliminates merging at all but the largest level of LSM-tree. The motivation is that merging at these smaller levels significantly increases update cost while yielding a comparatively insignificant improvement for point lookups, long range lookups, and space-amplification. Relative to leveling, we show that Lazy Leveling (1) improves the cost complexity of updates, (2) maintains the same complexity for point lookups, long range lookups, and space-amplification, and (3) provides competitive performance for short range lookups. We summarize the structure and performance characteristics of Lazy Leveling in Figure 6, and we discuss this figure in detail in the rest of the section.

Basic Structure. The top part of Figure 6 illustrates the structure of Lazy Leveling and compares it to tiering and leveling. Lazy leveling at its core is a hybrid of leveling and tiering: it applies leveling at the largest level and tiering at all other levels. As a result, the number of runs at the largest level is 1 and the number of runs at all other levels is at most $T-1$ (i.e., a merge operation takes place when the T^{th} run arrives).

Bloom Filters Allocation. Next, we show how to keep the cost complexity of point lookups fixed despite having more runs to probe at smaller levels. We do this by optimizing the memory budget among the Bloom filters across different levels. We start by modeling point lookup cost and the filters' overall main memory footprint with respect to the FPRs.

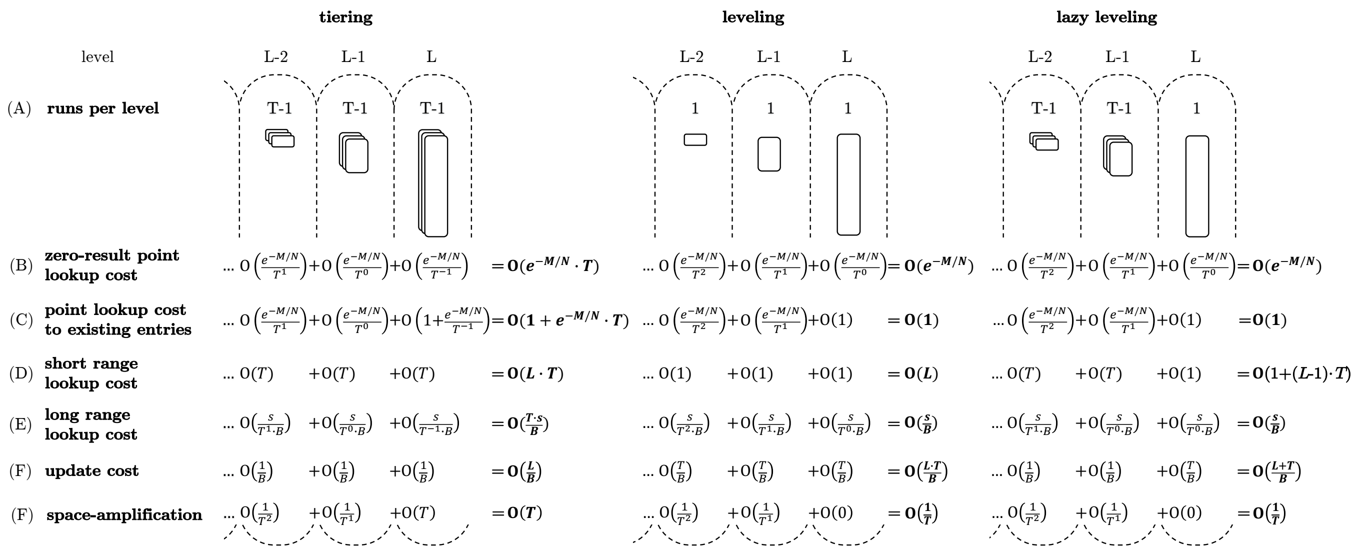

tiering
leveling
lazy leveling
level
L-2
L-1
L
(A) runs per level
T-1 T-1 T-1
1 1 1
T-1 T-1 1
(B) zero-result point lookup cost
$\dots O\left(\frac{e^{-M/N}}{T^1}\right) + O\left(\frac{e^{-M/N}}{T^0}\right) + O\left(\frac{e^{-M/N}}{T^{-1}}\right) = \mathbf{O}(e^{-M/N} \cdot T)$
$\dots O\left(\frac{e^{-M/N}}{T^2}\right) + O\left(\frac{e^{-M/N}}{T^1}\right) + O\left(\frac{e^{-M/N}}{T^0}\right) = \mathbf{O}(e^{-M/N})$
$\dots O\left(\frac{e^{-M/N}}{T^2}\right) + O\left(\frac{e^{-M/N}}{T^1}\right) + O\left(\frac{e^{-M/N}}{T^0}\right) = \mathbf{O}(e^{-M/N})$
(C) point lookup cost to existing entries
$\dots O\left(\frac{e^{-M/N}}{T^1}\right) + O\left(\frac{e^{-M/N}}{T^0}\right) + O\left(1 + \frac{e^{-M/N}}{T^{-1}}\right) = \mathbf{O}(1 + e^{-M/N} \cdot T)$
$\dots O\left(\frac{e^{-M/N}}{T^2}\right) + O\left(\frac{e^{-M/N}}{T^1}\right) + O(1) = \mathbf{O}(1)$
$\dots O\left(\frac{e^{-M/N}}{T^2}\right) + O\left(\frac{e^{-M/N}}{T^1}\right) + O(1) = \mathbf{O}(1)$
(D) short range lookup cost
$\dots O(T) + O(T) + O(T) = \mathbf{O}(L \cdot T)$
$\dots O(1) + O(1) + O(1) = \mathbf{O}(L)$
$\dots O(T) + O(T) + O(1) = \mathbf{O}(1 + (L\text{-}1) \cdot T)$
(E) long range lookup cost
$\dots O\left(\frac{s}{T^1 \cdot B}\right) + O\left(\frac{s}{T^0 \cdot B}\right) + O\left(\frac{s}{T^{-1} \cdot B}\right) = \mathbf{O}\left(\frac{T \cdot s}{B}\right)$
$\dots O\left(\frac{s}{T^2 \cdot B}\right) + O\left(\frac{s}{T^1 \cdot B}\right) + O\left(\frac{s}{T^0 \cdot B}\right) = \mathbf{O}\left(\frac{s}{B}\right)$
$\dots O\left(\frac{s}{T^1 \cdot B}\right) + O\left(\frac{s}{T^0 \cdot B}\right) + O\left(\frac{s}{T^0 \cdot B}\right) = \mathbf{O}\left(\frac{s}{B}\right)$
(F) update cost
$\dots O\left(\frac{1}{B}\right) + O\left(\frac{1}{B}\right) + O\left(\frac{1}{B}\right) = \mathbf{O}\left(\frac{L}{B}\right)$
$\dots O\left(\frac{T}{B}\right) + O\left(\frac{T}{B}\right) + O\left(\frac{T}{B}\right) = \mathbf{O}\left(\frac{L \cdot T}{B}\right)$
$\dots O\left(\frac{1}{B}\right) + O\left(\frac{1}{B}\right) + O\left(\frac{T}{B}\right) = \mathbf{O}\left(\frac{L+T}{B}\right)$
(F) space-amplification
$\dots O\left(\frac{1}{T^2}\right) + O\left(\frac{1}{T^1}\right) + O(T) = \mathbf{O}(T)$
$\dots O\left(\frac{1}{T^2}\right) + O\left(\frac{1}{T^1}\right) + O(0) = \mathbf{O}\left(\frac{1}{T}\right)$
$\dots O\left(\frac{1}{T^2}\right) + O\left(\frac{1}{T^1}\right) + O(0) = \mathbf{O}\left(\frac{1}{T}\right)$

Figure 6: A cost breakdown of updates, lookups, and space-amplification with different merge policies. Lazy Leveling achieves the same space-amplification, point lookup cost, and long range lookup cost as leveling while improving update cost and providing a slightly worse yet competitive short range lookup cost .

The worst-case expected number of wasted I/Os per lookup is issued by a zero-result point lookup and is equal to the sum of false positive rates across every run's Bloom filters. We model this cost for Lazy Leveling in Equation 3. The additive term p_L corresponds to the FPR for the single run at Level L, and the other term sums up the products of FPRs and number of runs at Levels 1 to $L - 1$.

$$R = p_L + (T - 1) \cdot \sum_{i=1}^{L-1} p_i \text{ where } 0 < p_i < 1 \qquad (3)$$

Next, we model the memory footprint M_i for the Bloom filters at Level i with respect to the number of entries N_i and the FPR p_i at that level. We do this by rearranging Equation 2 in terms of bits and applying it to each level. Since the filters at any given level all have the same FPR, we can directly apply this equation regardless of the numbers of runs at a level. The result is $M_i = -N_i \cdot \frac{\ln(p_i)}{\ln(2)^2}$. Next, we express N_i more generally as the product of the capacity at the largest level $N \cdot \frac{T-1}{T}$ and a discounting factor to adjust for the capacity at Level i: $\frac{1}{T^{L-i}}$. We then sum up the memory footprint across all levels to get the overall memory footprint M. The result is Equation 4.

$$M = -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \sum_{i=1}^{L} \frac{\ln(p_i)}{T^{L-i}} \qquad (4)$$

We now optimize Equations 3 and 4 with respect to each other to find the FPRs that minimize point lookup cost R with respect to a given memory budget M. To do so, we use the method of Lagrange multipliers. The full derivation is in Appendix A. The result is Equation 5.

$$p_i = \begin{cases} R \cdot \frac{T-1}{T}, & \text{for } i = L \\ R \cdot \frac{1}{T^{L-i+1}}, & \text{for } 1 \le i < L \end{cases} \qquad (5)$$

Zero-Result Point Lookups. Next, we analyze the cost of zero-result point lookups R with Lazy Leveling. We plug the optimal FPRs from Equation 5 into Equation 4, simplify into closed-form, and rearrange in terms of R. The complete derivation is in Appendix B. The result is Equation 6.

$$R = e^{-\frac{M}{N} \cdot \ln(2)^2} \cdot \frac{T^{\frac{T}{T-1}}}{(T-1)^{\frac{T-1}{T}}} \qquad (6)$$

This equation allows to quickly find the optimal FPRs with respect to a given memory budget M by plugging in the corresponding value of R from Equation 6 into Equation 5.

To analyze the complexity of zero-result point lookups, we observe that the multiplicative term at the right-hand side of Equation 6 is a small constant for any value of T. Therefore, the cost complexity is $O(e^{-M/N})$, the same as with leveling despite having eliminated most merge operations.

Memory Requirement. As the number of entries N grows relative to the memory budget M, the FPRs increase and eventually converge to one (starting from larger to smaller levels because the FPR at larger levels is higher). We identify the ratio of bits per entry M/N at which point the FPR at Level L converges to one by plugging in one for p_L in Equation 5, plugging the corresponding value of R into Equation 6, and rearranging in terms of $\frac{M}{N}$.

$$\text{threshold for } \frac{M}{N} = \frac{1}{\ln(2)^2} \cdot \left(\frac{\ln(T)}{T-1} + \frac{\ln(T-1)}{T} \right) \qquad (7)$$

Equation 7 has global maximum of $M/N = 1.62$ bits per entry (which occurs when T is set to 3). For mainstream key-value stores used for server applications, the default ratio is an order of magnitude larger, typically 10 [28, 32, 48] or 16 [52], and so the FPRs are all lower than one. For systems with less than 1.62 bits per entry (e.g., mobile devices or sensors), we adapt Lazy Leveling and its analysis in Appendix C by merging more at larger levels.

Point Lookups for Existing Entries. The worst-case point lookup cost to an existing entry occurs when the target key is at Level L. The expected I/O cost comprises one I/O to Level L plus the sum of FPRs across all other levels (i.e., $R - p_L$) and is given in Equation 8. The cost complexity is $O(1)$ I/Os because the term $R - p_L$ is always

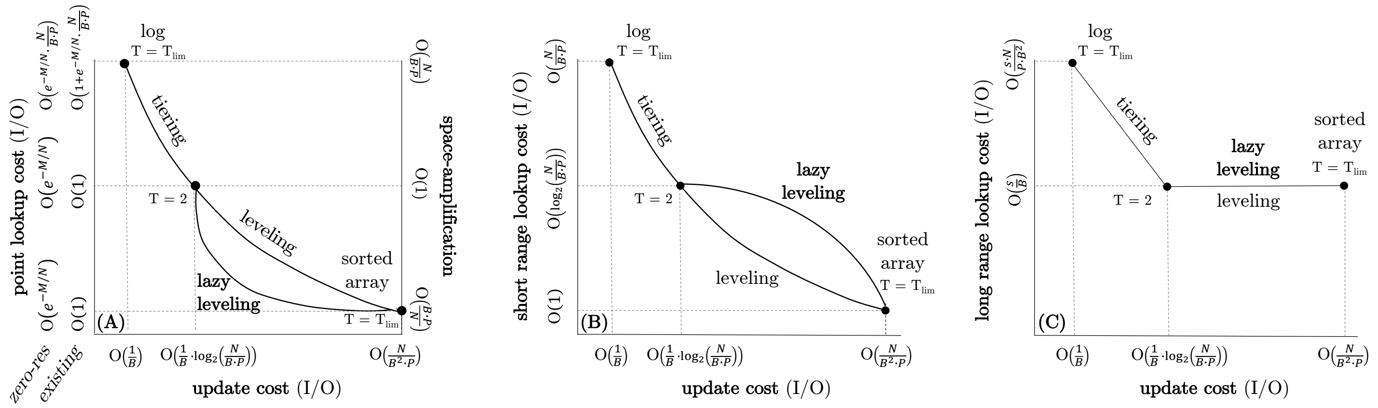


Figure 7: Lazy Leveling offers better trade-offs between updates and point lookups (Part A), worse trade-offs between updates and short range lookups (Part B), and the same trade-offs between updates and long range lookups (Part C).

less than 1 as long as the memory requirement in Equation 7 holds.

$$V = 1 + R - p_L \tag{8}$$

Range Lookups. A short range lookup issues at most $O(T)$ I/Os to each of the first $L-1$ levels and one I/O to the largest level, and so the cost complexity is $O(1 + (L-1) \cdot T)$ I/Os. Note that this expression initially increases as T increases, but as T approaches its limiting value of T_{lim} this term converges to 1 as the additive term $(L-1) \cdot T$ on the right-hand size becomes zero (i.e., at this point Lazy Leveling degenerates into a sorted array).

A long range lookup is dominated by sequential access to Level L because it contains exponentially more entries than all other levels. The cost is $O(\frac{s}{B})$ I/Os, where s is the size of the target key range relative to the size of the existing key space. This is the same as with leveling despite having eliminated most merge operations.

Updates. An updated entry with Lazy Leveling participates in $O(1)$ merge operations per level across Levels 1 to $L-1$ and in $O(T)$ merge operations at Level L. The overall number of merge operations per entry is therefore $O(L+T)$, and we divide it by the block size B to get the cost for a single update: $O(\frac{L+T}{B})$. This is an improvement over the worst-case cost with leveling.

Space-Amplification. In the worst case, every entry at Levels 1 to $L-1$ is an update to an existing entry at Level L. Since the fraction of entries at Levels 1 to $L-1$ is $\frac{1}{T}$ of the overall number of entries, space-amplification is at most $O(\frac{1}{T})$. This is the same bound as with leveling despite having eliminated most merge operations.

Limits. Figure 7 compares the behaviors of the different merge policies as we vary the size ratio T for each policy from 2 to its limit of T_{lim} (i.e., at which point the number of levels drops to one). Firstly, we observe that these policies converge in terms of performance characteristics when the size ratio T is set to 2 because at this point their behaviors become identical: the number of levels is the same and a merge operation occurs at every level when the second run arrives. Secondly, Part (A) of Figure 7 shows that the improvement that Lazy Leveling achieves for update cost relative to leveling can be traded for point lookup cost by increasing the size ratio. This generates a new trade-off curve between update cost and point lookup cost that dominates leveling, and converges with it again as T approaches T_{lim} (i.e., at which point both merge policies degenerate into a sorted array). Parts (B) shows that the cost of small range lookups is competitive, and part (C) shows that this cost difference becomes negligible as the target range grows.

Lesson: No Single Merge Policy Rules. Our analysis in figure Figure 7 shows that no single design dominates the others universally. Lazy leveling is best for combined workloads consisting of updates, point lookups and long range lookups, whereas tiering and leveling are best for workloads comprising mostly updates or mostly lookups, respectively. In the rest of the paper, we take steps towards a unified system that adapts across these designs depending on the application scenario.

4.2 Fluid LSM-Tree

To be able to strike all possible trade-offs for different workloads, we next introduce Fluid LSM-tree, a generalization of LSM-tree that enables switching and combining merge policies. It does this by controlling the frequency of merge operations separately for the largest level and for all other levels.

Basic Structure. Figure 8 illustrates the basic structure of Fluid LSM-tree. There are at most Z runs at the largest level and at most K runs at each of the smaller levels. To maintain these bounds, every Level i has an *active run* into which we merge incoming runs from Level $i-1$. Each active run has a size threshold with respect to the capacity of its level: $\frac{T}{K}$ percent for Levels 1 to $L-1$ and $\frac{T}{Z}$ percent for Level L. When an active run reaches this threshold, we start a new active run at that level. Ultimately when a level is at capacity, all runs in it get merged and flushed down to the next level.

Parameterization. The bounds K and Z are used as tuning parameters that enable Fluid LSM-tree to assume the behaviors of different merge policies.

- $K = 1$ and $Z = 1$ give leveling.
- $K = T - 1$ and $Z = T - 1$ give tiering.
- $K = T - 1$ and $Z = 1$ give Lazy Leveling.

Fluid LSM-tree can transition from Lazy Leveling to tiering by merging less frequently at the largest level by increasing Z, or it can transition to leveling by merging more frequently at all other levels by decreasing K. Fluid LSM-tree spans all possible trade-offs along and between the curves in Figure 7.

Bloom Filters Allocation. Next, we derive the optimal FPRs that minimize point lookup cost with respect to the parameters K and

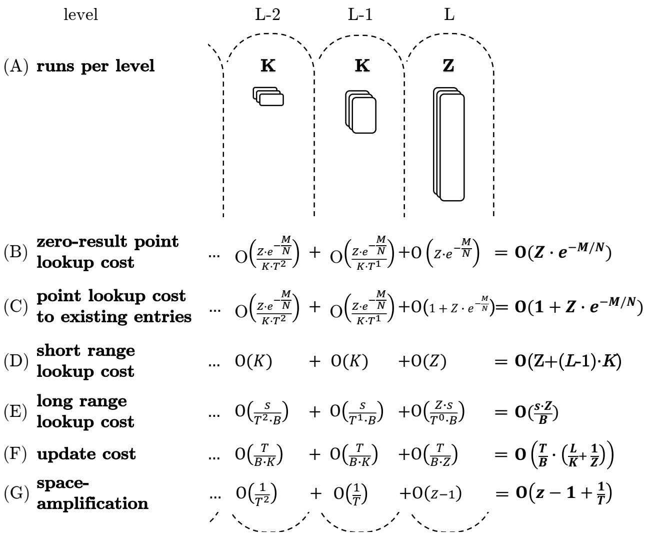

level
L-2
L-1
L
(A) runs per level
K
K
Z
(B) zero-result point lookup cost
... $O\left(\frac{Z\cdot e^{-\frac{M}{N}}}{K\cdot T^2}\right) + O\left(\frac{Z\cdot e^{-\frac{M}{N}}}{K\cdot T^1}\right) + O\left(Z\cdot e^{-\frac{M}{N}}\right) = \mathbf{O(Z\cdot e^{-M/N})}$
(C) point lookup cost to existing entries
... $O\left(\frac{Z\cdot e^{-\frac{M}{N}}}{K\cdot T^2}\right) + O\left(\frac{Z\cdot e^{-\frac{M}{N}}}{K\cdot T^1}\right) + O(1+Z\cdot e^{-\frac{M}{N}}) = \mathbf{O(1+Z\cdot e^{-M/N})}$
(D) short range lookup cost
... $O(K) + O(K) + O(Z) = \mathbf{O(Z+(L\text{-}1)\cdot K)}$
(E) long range lookup cost
... $O(\frac{s}{T^2\cdot B}) + O(\frac{s}{T^1\cdot B}) + O(\frac{Z\cdot s}{T^0\cdot B}) = \mathbf{O(\frac{s\cdot Z}{B})}$
(F) update cost
... $O(\frac{T}{B\cdot K}) + O(\frac{T}{B\cdot K}) + O(\frac{T}{B\cdot Z}) = \mathbf{O\left(\frac{T}{B}\cdot(\frac{L}{K}+\frac{1}{Z})\right)}$
(G) space-amplification
... $O(\frac{1}{T^2}) + O(\frac{1}{T}) + O(Z-1) = \mathbf{O(z-1+\frac{1}{T})}$

Figure 8: A cost breakdown of updates, lookups, and space-amplification with Fluid LSM-tree.

Z. The derivation is in Appendix A and the result is Equation 9.

$$p_i = \begin{cases} \frac{R}{Z}\cdot\frac{T-1}{T}, & \text{for } i = L \\ \frac{R}{K}\cdot\frac{T-1}{T}\cdot\frac{1}{T^{L-i}}, & \text{for } 1 \le i < L \end{cases} \tag{9}$$

Equation 9 generalizes the optimal Bloom filters allocation strategy in Monkey [22] across a significantly wider design space, which now, in addition to tiering and leveling, also includes Lazy Leveling as well as custom merge policies with any parameter values for K and Z. Next, we model and map the new space-time trade-offs that this expanded design space offers.

Zero-Result Point Lookups. We model the cost of zero-result point lookups by plugging the generalized optimal FPRs in Equation 9 into Equation 4, simplifying into closed-form, and rearranging in terms of R. The derivation is in Appendix B, experimental validation is in Appendix I, and the result is Equation 10. The generalized complexity is $O(Z\cdot e^{-M/N})$ I/Os.

$$R = e^{-\frac{M}{N}\cdot\ln(2)^2}\cdot Z^{\frac{T-1}{T}}\cdot K^{\frac{1}{T}}\cdot\frac{T^{\frac{T}{T-1}}}{T-1} \tag{10}$$

Point Lookups for Existing Entries. The worst-case lookup cost to an existing key occurs when the target key is at the oldest run at the largest level. The expected I/O cost is one I/O to this target run plus the sum of FPRs across all other runs. We use Equation 8 to model this, and we plug in Equation 10 for R and Equation 9 for p_L. The generalized complexity is $O(1 + Z\cdot e^{-M/N})$.

Memory Requirement. In Appendix C, we derive the memory requirement M/N that guarantees that FPRs across all Levels are lower than one. The generalized result is 1.62 bits per entry as in the last subsection, which is well below the default ratio in mainstream systems. In Appendix C, we show how to adapt Fluid LSM-tree to extremely low-memory environments.

Range Lookups. A short range lookup issues at most K I/Os per level to the smaller $L-1$ Levels and at most Z I/Os to the largest level for a total of $Z+K\cdot(L-1)$ random I/Os and a cost complexity of $O(Z+K\cdot(L-1))$. A long range lookup continues with a sequential scan to the relevant key range at each run issuing at least $\frac{s}{B}$ sequential I/Os, where s is the number of unique entries in the target key range. To account for obsolete entries, the number of sequential I/Os is amplified by a factor of $1+\frac{1}{T}$ for updated entries at Levels 1 to $L-1$ and Z for updated entries at Level L, which we model together as $Z+\frac{1}{T}$. The sequential scan cost is therefore at most $\frac{s}{B}\cdot(Z+\frac{1}{T})$ I/Os with a complexity of $O(\frac{s\cdot Z}{B})$ I/Os. The generalized range lookup cost is given in Equation 11 as the sum of costs for short and long range lookups weighted by the constant μ, the amount of which sequential access is faster than random access on a given storage devices (e.g., disk).

$$Q = K\cdot(L-1) + Z + \frac{1}{\mu}\cdot\frac{s}{B}\cdot\left(Z+\frac{1}{T}\right) \tag{11}$$

Updates. In the worst case, an entry participates in $O(\frac{T}{K})$ merge operations within an active run across each of Levels 1 to $L-1$, and in $O(\frac{T}{Z})$ merge operations within the active run at Level L. The overall update cost is the sum of these terms across all levels divided by the block size: $O\left(\frac{T}{B}\cdot\left(\frac{L}{K}+\frac{1}{Z}\right)\right)$. We model this cost more precisely using arithmetic series to obtain Equation 12, which we validate in Appendix I. We divide by the constant μ since the cost of updates is incurred through sequential merge operations, and we introduce an additional constant ϕ to account for the property of some storage devices that writes are more expensive than reads (e.g., flash).

$$W = \frac{\phi}{\mu\cdot B}\cdot\left(\frac{T-1}{K+1}\cdot(L-1) + \frac{T-1}{Z+1}\right) \tag{12}$$

Space-Amplification. Levels 1 to $L-1$ contain a fraction of $\frac{1}{T}$ of the dataset, and so they may render up to this fraction of entries obsolete at the largest level. In Level L, at most $Z-1$ of the runs may be completely filled with obsolete entries. We model space-amplification as the sum of these terms in Equation 13.

$$amp = Z - 1 + \frac{1}{T} \tag{13}$$

Mapping the Design Space. Figure 9 is an instance of conceptual Figure 7 that uses our cost models to map the different trade-offs with Fluid LSM-tree. We generate Part (A) of Figure 9 by plotting point lookup cost R in Equation 10 against update cost W in Equation 12. We generate Parts (B) and (C) for short and long range lookups by plotting Q in Equation 11 against update cost W in Equation 12 for selectivities s of 10^{-7} and 10^{-6}, respectively. We leave an evaluation of space-amplification for the experimental analysis. The configuration is fixed to a 1TB dataset with 128 byte entries, 4KB storage blocks, and overall 10 bits per entry across the filters[5]. We generate the curves for leveling, tiering, and Lazy Leveling by using their corresponding fixed values for the parameters K and Z, and varying the size ratio T. The circle indicates the convergence point of all three merge policies where the size ratio T is set to two. The squares indicate a size ratio of ten, which most mainstream key-value stores use by default in practice [28, 32], to enable comparison of corresponding points across the three sub-figures. The figure also illustrates two transition curves labeled Trans1 and Trans2, which demonstrate how Fluid LSM-tree transitions fluidly across designs thereby achieving trade-offs that would not have been possible using a fixed merge policy.

[5] We set $N = 2^{33}$, $E = 2^7$, $B = 2^5$, and $M = 10\cdot 2^{33}$.

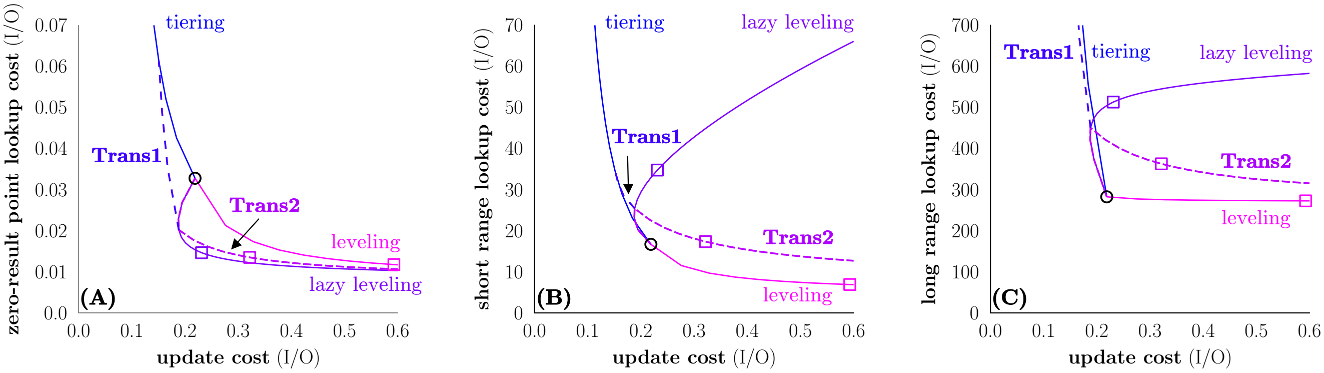


Figure 9: Fluidly shifting between Lazy Leveling, leveling, and tiering.

Transition 1: Lazy Leveling to Tiering. We observe in Part (A) of Figure 9 that the curve for Lazy Leveling has an inflection point beyond which decreasing the size ratio degrades update cost. The reason is that update cost is $O(\frac{L+T}{B})$, and as we decrease T the value of L grows and comes to dominate T. In this example, the inflection point occurs when the size ratio T is set to 5. We generate the curve labeled Transition 1 (Trans1) by fixing T to the inflection point value and instead varying Z from 1 to $T-1$ (4 in this example). The resulting curve dominates both Lazy Leveling and tiering for this part of the design space until it converges with tiering. Thus, Transition 1 enables optimal trade-offs between point lookup cost and update cost as we transition between Lazy Leveling and tiering to optimize more for point lookups or updates, respectively.

Transition 2: Lazy Leveling to Leveling. In order to achieve more competitive range lookup costs with Lazy Leveling, we introduce Transition 2. The idea is to vary K, the bound on runs at Levels 1 to $L-1$, between 1 and $T-1$ to fluidly transition between Lazy Leveling and leveling. In Figure 9 we generate the curve labeled Trans2 by fixing K to 4 and varying T. Part (A) shows that this enables navigating a trade-off curve similar to Lazy Leveling, and parts (B) and (C) show that Trans2 achieves nearly the same range lookup cost as with leveling. Thus, Transition 2 enables fine control over how much we optimize for short range lookups.

Problem: Finding the Best Tuning. The space of optimal space-time trade-offs is delineated by leveling, Lazy Leveling, and tiering, and we can reach many other trade-offs in-between by co-tuning the parameters K, Z and T. The goal is to co-tune these parameters to strike the best trade-off for a particular application. The challenge is finding the optimal configuration as there are many different combinations for these parameters[6].

4.3 Dostoevsky

We now introduce Dostoevsky to find and adapt to the best tuning of Fluid LSM-tree subject to a constraint on space-amplification. Dostoevsky models and optimizes throughput with respect to update cost W in Equation 12, zero-result point lookup cost R in Equation 10, non-zero result point lookup cost V in Equation 8, and range lookup cost Q in Equation 11. It monitors the proportion of these operations in the workload and weights their costs using coefficients w, r, v, and q, respectively. We multiply this weighted cost by the time to read a block from storage Ω and taking the inverse to obtain the weighted worst-case throughput τ.

$$\tau = \Omega^{-1} \cdot (w \cdot W + r \cdot R + v \cdot V + q \cdot Q)^{-1} \tag{14}$$

Dostoevsky maximizes Equation 14 by iterating over different values of the parameters T, K, and Z. It prunes the search space using two insights. The first is that LSM-tree has at most L_{max} levels, each of which has a corresponding size ratio T, and so there are only $\lceil \log_2(\frac{N}{P \cdot B}) \rceil$ meaningful values of T to test. The second insight is that the lookup costs R, Q and V increase monotonically with respect to K and Z, whereas update cost W decreases monotonically with respect to them. As a result, Equation 14 is convex with respect to both K and Z, and so we can divide and conquer their value spaces and converge to the optimum with logarithmic runtime complexity. Overall, auto-tuning takes $O(log_2(\frac{N}{B \cdot P})^3)$ iterations as each parameter contributes one multiplicative log factor to runtime. To satisfy a given constraint on space-amplification, we ignore tunings for which Equation 13 is above the constraint. Since we iterate over a closed-form model, execution takes a fraction of a second, making it possible to find the optimal tuning at runtime without affecting overall system performance. We invoke auto-tuning between time windows consisting of X buffer flushes (16 in our implementation). A more detailed description of the adaptation workflow and the transition overheads is given in Appendix G.

5 EVALUATION

We evaluate Dostoevsky across a range of workloads and show that it dominates existing designs in terms of performance and space-amplification.

Experimental Infrastructure. We use a machine with a RAID of 500GB 7200RPM disks, 32GB DDR4 main memory, 4 2.7 GHz cores with 8MB L3 caches, running 64-bit Ubuntu 16.04 LTS on an ext4 partition with journaling turned off.

Implementation. We implemented Dostoevsky on RocksDB [28], an LSM-tree based key-value store that is widely used in industry [8, 27]. RocksDB only supports leveling and assigns fixed FPR to Bloom filters across all levels. We optimized the Bloom filters allocation by embedding Equation 9 within the code. We then implemented Fluid LSM-tree using a RocksDB API that enables listening to internal events and scheduling merge operations using custom logic. We implemented auto-tuning by measuring the proportion of different operations in the workload during time windows and feeding them to Equation 14. We control the main memory budget by enabling

[6] The number of combinations is $O((\frac{N}{B \cdot P})^3)$. The reason is that the maximum value of T is $T_{lim} = \frac{N}{B \cdot P}$, and K and Z can both be tuned to anything between 1 and $T-1$.

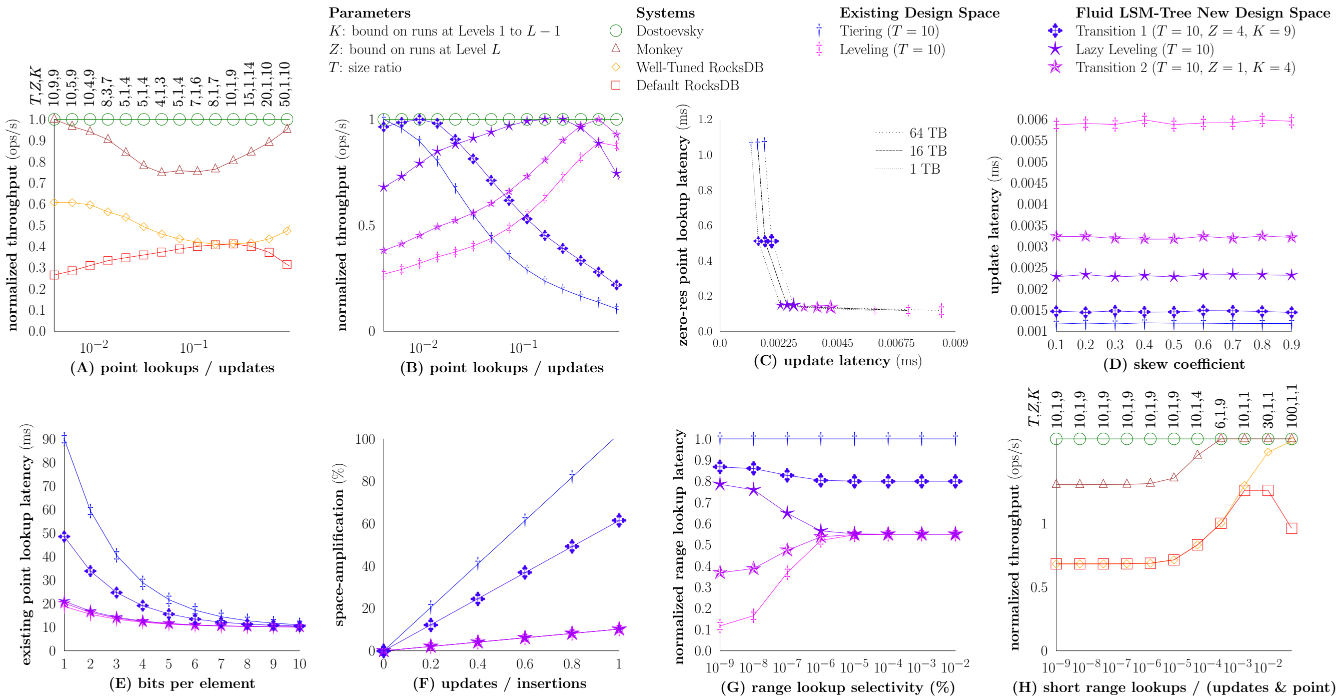


Figure 10: Dostoevsky dominates existing designs by being able to strike better space-time trade-offs.

direct IO, and as key-value stores in industry [28, 32] we set the buffer size to 2 MB, the number of bits per entry across all Bloom filters to 10, the density of fence pointers to one per 32KB block, and the block cache size to be 10% of the dataset size in each experiment.

Metrics. To measure the latency of updates and lookups, we monitor the amount of time spent by the disk processing lookups and merge operations, respectively, and we divide these measures by the number of lookups and updates in each trial. We measure space-amplification by dividing the raw data size by the volume of unique entries inserted. Each point in the resulting figures represents an average of three experimental trials.

Baselines. We compare Dostoevsky to RocksDB [28] and to Monkey [22], which represent the state-of-the-art designs in industry and research, respectively. For RocksDB, we compare against the default tuning (leveling with a size ratio of 10) and against our own tuning of the size ratio to maximize throughput. We experimented with Monkey as a subset of Dostoevsky with only tiering and leveling in its design space. To enable a finer study of the design space, we also compare among five fixed merge policies: Leveling, Tiering, and Lazy Leveling all with a size ratio of 10 and optimized Bloom filters, as well as Transitions 1 and 2 from Section 4.2, which represent combinations of these policies. Write-ahead-logging is enabled across all systems. Since all of these baselines run on top of the RocksDB infrastructure, any variations in performance or space are purely due to differences in data structure design and tuning.

Experimental Setup. The default setup involves inserting 1TB of data to an empty database. Every entry is 1KB (the key is 128 bytes and the attached value is 896 bytes, which is common in practice [8]). We compare the designs under worst-case conditions using by default a worst-case workload consisting of uniformly randomly distributed operations, and we vary the proportion of updates and lookups to test the extent to which we can optimize throughput given the contention between their costs. This is in line with the Yahoo Cloud Service Benchmark [21], which is commonly used to evaluate key-value stores today [39, 45, 47, 48, 53]. For generality, we also compare the designs across workload skews. To ensure experimental control, we begin each trial from a fresh clone, as we discuss in more detail in Appendix F.

Dostoevsky Dominates Existing Systems. In Figure 10 (A), we compare Dostoevsky to RocksDB and Monkey as we increase the proportion of zero-result point lookups to updates in the workload from 0.5% to 95% (the x-axis is log scale and the y-axis is normalized to enable a clear comparison). Both versions of RocksDB perform poorly because (1) they allocate memory uniformly across the Bloom filters thereby incurring more I/Os due to false positives, and (2) they are restricted to using only leveling. Monkey improves on them by optimizing the Bloom filters allocation and also including tiering in its design space to be able to optimize more for updates. Dostoevsky dominates these systems because Fluid LSM-tree and Lazy Leveling enable better cost trade-offs for combined workloads. The numbers above the figure show the tunings of Dostoevsky used for each workload. These tunings are all unique to the Lazy Leveling and Fluid LSM-tree design spaces, except at the edges where the workload consists of mostly one type of operation.

No Single Merge Policy Rules. In Figure 10 (B), we repeat the same experiment while comparing the different merge policies. As the proportion of lookups increases, the best fixed policy changes from tiering to Transition 1, then to Lazy Leveling, then to Transition 2, and finally to leveling. A fixed merge policy is only best for one particular proportion of updates and lookups. We normalize the y-axis based on the throughput that Dostoevsky achieves for each workload (using the same tunings at the top of Part (A)). Dostoevsky dominates all fixed policies by encompassing all of them and fluidly transitioning among them.

Dostoevsky is More Scalable. In Figure 10 (C), we compare Lazy Leveling to the other merge policies as we increase the data size. Point lookup cost does not increase as the data size grows because we scale the Bloom filters' memory budget in proportion to the data size. The first takeaway is that Lazy Leveling achieves the same point lookup cost as leveling while significantly reducing update cost by eliminating most merge operations. The second takeaway is that update cost with Lazy Leveling scales at a slower rate than with leveling as the data size increases because it merges greedily only at the largest level even as the number of levels grows. As a result, Dostoevsky offers increasingly better performance relative to state-of-the-art designs as the data size grows.

Robust Update Improvement Across Skews. In Figure 10 (D), we evaluate the different fixed merge policies under a skewed update pattern. We vary a skew coefficient whereby a fraction of c of all updates target a fraction of $1 - c$ of the most recently updated entries. The flat lines show that update cost is largely insensitive to skew with all merge policies. The reason is that levels in LSM-tree grow exponentially in size, and so even an update to the most recent 10% of the data traverses all levels rather than getting eliminated prematurely while getting merged across smaller levels. Hence, the improved update cost offered by Lazy Leveling and by Dostoevsky by extension is robust across a wide range of temporal update skews. We experiment with point lookup skews in Appendix H.

Robust Performance Across Memory Budgets. In Figure 10 (E), we vary the memory budget allocated to the Bloom filters and we measure point lookup latency to existing entries. Even with as little as one bit per entry, Lazy Leveling achieves comparable latency to leveling, and so Dostoevsky achieves robust point lookup performance across a wide range of memory budgets.

Lower Space-Amplification. In Figure 10 (F), we vary the proportion of updates of existing entries to insertions of new entries, and we measure space-amplification. As the proportion of updates increases, there are more obsolete entries at larger levels, and so space-amplification increases. We observe that space-amplification for Lazy Leveling and Leveling increase at the same rate, despite eliminating most merge operations. As a result, Dostoevsky is able to achieve a given bound on space-amplification while paying a lower toll in terms of update cost due to merging.

Domination with Range Lookups. Figure 10 (G) compares range lookup costs as selectivity increases. The first takeaway is that for low selectivities, Transition 2 achieves a competitive short range lookup cost with leveling while achieving comparable bounds on point lookups and space-amplification (as showed in Figures 10 (C), (E) and (F), respectively). The second takeaway is that as selectivity increases Lazy Leveling becomes increasingly competitive. We demonstrate how Dostoevsky uses these designs in Figure 10 (H), where we increase the proportion of short range lookups in the workload while keeping a fixed ratio of five updates per point lookup. The best fixed merge policy changes from Lazy Leveling to Transition 2 to Leveling as the proportion of range lookups increases. Dostoevsky transitions among these policies thereby maximizing throughput and dominating existing systems.

6 RELATED WORK

Key-Value Stores in Industry. Mainstream key-value stores in industry [6, 19, 28, 31, 32, 34, 48, 52] have the same architecture as Dostoevsky: they persist data using LSM-tree and speed up point lookups using Bloom filters. However, they are tuned suboptimally over a small subset of the design space. LevelDB [32] hard codes the size ratio to 10 and the merge policy to leveling. RocksDB [28] supports leveling but not tiering. bLSM [48] restricts the number of levels to 3 by merging more aggressively as data grows. None of these systems optimize memory allocation across the Bloom filters to minimize point lookup cost. Overall, it is hard to tune these systems even within their limited design spaces as the impact of any existing tuning parameter on performance is unclear.

Key-Value Stores in Research. Key-value stores in research expand the trade-offs that are possible to achieve and navigate. Some designs use fractional cascading [13, 35] rather than fence pointers thereby trading lookup cost for memory. B^{ϵ}-tree and variants [2, 15, 42] use buffered in-place updates among linked nodes instead of fence pointers thereby trading memory for space and range lookup cost, as nodes are padded and non-contiguous. Other designs [4, 11, 24, 25] use circular logs in storage and index them using a hash table in memory thereby trading space for update cost. Recent work [36] optimizes the size ratio of leveled LSM-tree for update skew. PebblesDB [45] merges more frequently across frequently accessed parts of the key space. Monkey [22] allocates main memory among the Bloom filters of LSM-tree to minimize point lookup cost, and it transitions between leveling and tiering to maximize throughput.

Dostoevsky. Our insight is that the recent Bloom filters optimization in Monkey [22] offers a new avenue for optimizing update cost by relaxing merging at lower levels while keeping point lookup cost fixed. We exploit this by introducing Lazy Leveling, which offers better scalability trade-offs among point lookups, long range lookups, updates, and space-amplification than we knew existed before. We further introduce Fluid LSM-Tree and Dostoevsky to generalize and be able to navigate the expanded design space.

Related Topics. In Appendix E, we discuss complementary techniques for reducing and scheduling merge operations. We also discuss specialized and in-memory key-value stores.

7 CONCLUSION

We show that merges in LSM-tree based key-value stores control an intrinsic trade-off among the costs of updates, point lookups, range lookups, and space-amplification, yet existing designs do not strike optimal trade-offs among these metrics. The reason is that merge operations do the same amount of maintenance work across all levels, yet point lookups, long range lookups, and space-amplification derive their cost mostly from the largest level. We introduce Dostoevsky, a key-value store that offers richer space-time trade-offs by merging as little as possible to achieve given bounds on lookup cost and space, and we show how to find the best trade-off for a particular application workload and hardware.

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This work is supported by the National Science Foundation under grant IIS-1452595.

REFERENCES

[1] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[3] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment*, 8(8):850–861, 2015.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 2009.

[5] Apache. Accumulo. *https://accumulo.apache.org/*, 2016.

[6] Apache. Cassandra. *http://cassandra.apache.org*, 2016.

[7] Apache. HBase. *http://hbase.apache.org/*, 2016.

[8] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.

[9] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 865–876, 2011.

[10] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *ACM Transactions on Database Systems (TODS)*, 40(1), 2015.

[11] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache Storage for the Next Billion. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 123–136, 2009.

[12] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *USENIX Annual Technical Conference*, 2017.

[13] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-Oblivious Streaming B-trees. In *Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 81–92, 2007.

[14] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] G. S. Brodal and R. Fagerberg. Lower Bounds for External Memory Dictionaries. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 546–554, 2003.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[17] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and Condie. Pregelix: Big (ger) graph analytics on a dataflow engine. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 161–172, 2014.

[18] Z. Cao, S. Chen, F. Li, M. Wang, and X. S. Wang. LogKV: Exploiting Key-Value Stores for Log Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[19] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[20] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: A Unified Cloud Object Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754, 2012.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, New York, NY, USA, 2010. ACM.

[22] N. Dayan, M. Athanassoulis, and S. Idreos. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 79–94, 2017.

[23] N. Dayan, P. Bonnet, and S. Idreos. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 327–342, 2016.

[24] B. Debnath, S. Sengupta, and J. Li. FlashStore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[25] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 25–36, 2011.

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[27] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing Space Amplification in RocksDB. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2017.

[28] Facebook. RocksDB. *https://github.com/facebook/rocksdb*, 2016.

[29] Facebook. MyRocks. *http://myrocks.io/*, 2017.

[30] B. Fitzpatrick and A. Vorobey. Memcached: a distributed memory object caching system, 2011.

[31] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling Concurrent Log-Structured Data Stores. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 32:1—-32:14, 2015.

[32] Google. LevelDB. *https://github.com/google/leveldb/*, 2016.

[33] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 16–25, 1997.

[34] A. Lakshman and P. Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[35] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[36] H. Lim, D. G. Andersen, and M. Kaminsky. Towards Accurate and Fast Evaluation of Multi-Stage Log-structured Designs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 149–166, 2016.

[37] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011.

[38] LinkedIn. Online reference. *http://www.project-voldemort.com*, 2016.

[39] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 133–148, 2016.

[40] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. *ACM/IEEE IPSN*, 2007.

[41] 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.

[42] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 537–550, 2016.

[43] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquin, and D. Kossmann. Fast Scans on Key-Value Stores. *Proceedings of the VLDB Endowment*, 10(11):1526–1537, 2017.

[44] J. Pitman. *Probability*. 1999.

[45] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. *Proceedings of the ACM Symposium on Operating Systems Principles*, 2017.

[46] Redis. Online reference. *http://redis.io/*.

[47] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proceedings of the VLDB Endowment*, 10(13):2037–2048, 2017.

[48] R. Sears and R. Ramakrishnan. bLSM: A General Purpose Log Structured Merge Tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–228, 2012.

[49] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building Workload-Independent Storage with VT-trees. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 17–30, 2013.

[50] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Serveys & Tutorials*, 14(1):131–155, 2012.

[51] R. Thonangi and J. Yang. On Log-Structured Merge for Solid-State Drives. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 683–694, 2017.

[52] WiredTiger. WiredTiger. *https://github.com/wiredtiger/wiredtiger*, 2016.

[53] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 71–82, 2015.

A OPTIMAL FALSE POSITIVE RATES

In this appendix, we derive the optimal false positive rates (FPRs) for Fluid LSM-tree that minimize the zero-result point lookup cost. This result can then be applied to any particular merge policy by plugging in the appropriate parameters K and Z. First, we model point lookup cost with respect to these parameters:

$$R = Z \cdot p_L + K \cdot \sum_{i=1}^{L-1} p_i \qquad (15)$$

Next, we use the method of Lagrange multipliers to find the FPRs that minimize Equation 4 subject to Equation 15 as a constraint. We first express these in the standard form:

	Tiering	Leveling	Lazy Leveling
zero-res point	$O(T \cdot Y)$	$O(Y)$	$O(Y)$
existing point	$O(T \cdot Y)$	$O(Y)$	$O(Y)$
short range	$O(T \cdot L)$	$O(L)$	$O(T \cdot L + T \cdot (1 - T))$
long range	$O(\frac{s \cdot T}{B})$	$O(\frac{s}{B})$	$O(\frac{s}{B})$
updates	$O(\frac{L}{B})$	$O(\frac{T \cdot L}{B}))$	$O(\frac{L+Y \cdot T}{B})$
space-amp	$O(T)$	$O(\frac{1}{T})$	$O(\frac{1}{T})$

Table 2: I/O analysis below the memory requirement.

$$\begin{aligned} g(p_L \ldots p_1, R, K, Z) &= Z \cdot p_L + K \cdot \sum_{i=1}^{L-1} (p_i) - R \\ y(p_L \ldots p_1, N, T) &= -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \sum_{i=1}^{L} \frac{\ln(p_i)}{T^{L-i}} \end{aligned} \tag{16}$$

We express the Lagrangian $\mathcal{L}$ in terms of these functions:

$$\mathcal{L} = y(p_L \ldots p_1, N, T) + \lambda \cdot g(p_L \ldots p_1, R, K, Z) \tag{17}$$

Next, we differentiate the Lagrangian with respect to each of the FPRs $p_1 \ldots p_L$, and we set every partial derivative to 0. We arrive at the following system of equations.

$$\begin{aligned} \frac{N}{\ln(2)^2 \cdot \lambda} &= P_L \cdot Z \\ \frac{N}{\ln(2)^2 \cdot \lambda} &= P_{L-i} \cdot K \cdot T^i \end{aligned} \tag{18}$$

We equate these equations to eliminate the constants.

$$Z \cdot P_L = P_{L-1} \cdot T \cdot K = P_{L-2} \cdot T^2 \cdot K = \ldots$$

We now express all of the optimal FPRs in terms of the optimal FPR for the largest level P_L.

$$P_{L-i} = \frac{P_L \cdot Z}{T^i \cdot K}$$

Next, we express Equation R in terms of only T and P_L by plugging these FPRs into Equation 4. We observe that R is now expressed in terms of a geometric series. We simplify it using the formula for the sum of a geometric series up to L elements.

$$\begin{aligned} R &= Z \cdot p_L + \frac{Z \cdot p_L}{T^1} + \frac{Z \cdot p_L}{T^2} + \ldots \\ R &= Z \cdot p_L \cdot \frac{\frac{1}{T^L} - \frac{1}{T}}{\frac{1}{T} - 1} \end{aligned} \qquad in \tag{19}$$

As L grows, this converges to the following.

$$R = Z \cdot p_L \cdot \frac{T}{T-1} \tag{20}$$

As a result, the optimal false positive rates are as follow.

$$p_i = \begin{cases} \frac{R}{Z} \cdot \frac{T-1}{T}, & \text{for } i = L \\ \frac{R}{K} \cdot \frac{T-1}{T} \cdot \frac{1}{T^{L-i}}, & \text{for } 1 \le i < L \end{cases} \tag{21}$$

To get the optimal FPR assignment for Lazy Leveling as in Equation 5, we plug 1 for Z and $T-1$ for K in Equation 21.

B CLOSED-FORM POINT LOOKUP COST

We now model zero-result point lookup cost in closed form by plugging the optimal FPRs in Equation 21 into Equation 4, simplify into closed-form by applying logairthmic operations and sums of series (i.e., geometric and arithmetico-geometric), and finally rearranging in terms of R, as follows.

$$\begin{aligned} M &= -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \left(\ln\left(\frac{R}{Z} \cdot \frac{T-1}{T} \right) + \sum_{i=1}^{L} \frac{1}{T^i} \ln\left(\frac{R}{K} \cdot \frac{T-1}{T} \cdot \frac{1}{T^i} \right) \right) \\ &= -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \left(\ln\left(\frac{R}{Z} \cdot \frac{T-1}{T} \right) + \ln\left(\left(\frac{(T-1) \cdot R}{T \cdot K} \right)^{\frac{1}{T} + \frac{1}{T^2} + \frac{1}{T^3} + \cdots} \cdot \left(\frac{1}{T} \right)^{\frac{1}{T^1} + \frac{2}{T^2} + \frac{3}{T^3} \cdots} \right) \right) \\ &= -\frac{N}{\ln(2)^2} \cdot \frac{T-1}{T} \cdot \ln\left(\frac{1}{Z} \cdot \left(\frac{(T-1) \cdot R}{T} \right)^{\frac{T}{(T-1)}} \cdot \left(\frac{1}{K} \right)^{\frac{1}{(T-1)}} \cdot \left(\frac{1}{T} \right)^{\frac{T}{(T-1)^2}} \right) \\ &= \frac{N}{\ln(2)^2} \cdot \ln\left(\frac{Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}}}{R} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \right) \\ R &= e^{-\frac{M}{N} \cdot \ln(2)^2} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} \end{aligned} \tag{22}$$

C MEMORY REQUIREMENT

We now generalize Fluid LSM-tree for any memory budget. We first identify the ratio of bits per element X at which the FPR at Level L converges to one. We plug in one for p_L in Equation 21, plug the corresponding value of R into Equation 10, and rearrange in terms of $\frac{M}{N}$. The result is as follows.

$$X = \frac{1}{\ln(2)^2} \cdot \left(\frac{\ln(T)}{T-1} + \frac{\ln(K) - \ln(Z)}{T} \right) \tag{23}$$

If the memory ratio M/N is lower than the threshold X, we denote Y as the number of levels from the largest down whose FPRs have converged to one. We solve for Y by discounting the number of entries at the largest Y levels: $X = \lfloor (\frac{M}{N} \cdot T^Y \rfloor)$. Rearranging in terms of Y gives Equation 24.

$$Y = \lfloor \log_T \left(\frac{N \cdot X}{M} \right) \rfloor \tag{24}$$

When Y is one or greater, Fluid LSM-tree applies a bound of Z runs on the number of runs at the largest Y levels. The optimal FPR assignment now only applies to a smaller version of the problem at the smaller $L - Y$ levels, according to Equation 25.

$$p_i = \begin{cases} 1, & \text{for } i > L - Y \\ \frac{R - Y \cdot Z}{Z} \cdot \frac{T-1}{T}, & \text{for } i = L - Y \\ \frac{R - Y \cdot Z}{K} \cdot \frac{T-1}{T} \cdot \frac{1}{T^{L-i}}, & \text{for } 1 \le i < L - Y \end{cases} \tag{25}$$

We adjust the cost models as follows. Zero-result point lookups now issue I/Os to the largest Y levels, and so we get Equation 26.

$$R = e^{-\frac{M}{N} \cdot \ln(2)^2 \cdot T^Y} \cdot Z^{\frac{T-1}{T}} \cdot K^{\frac{1}{T}} \cdot \frac{T^{\frac{T}{T-1}}}{T-1} + Y \cdot Z \tag{26}$$

Point lookups to existing entires now also access every run at the largest Y levels, and so the I/O cost is the same as for zero-result point lookups in Equation 26.

Range lookups issue Z I/Os to each of the largest Y levels, and so we get Equation 27.

$$Q = K \cdot (L - Y) + Z \cdot Y + \frac{s}{\mu \cdot B} \cdot \left(Z + \frac{1}{T} \right) \tag{27}$$

Updates now merge each entry in the largest Y levels $\frac{T}{Z}$ times per level, and so we get Equation 28.

$$W = \frac{\phi}{\mu \cdot B} \cdot \left(\frac{T-1}{K+1} \cdot (L - Y) + \frac{T-1}{Z+1} \cdot Y \right) \tag{28}$$

We summarize these results in Table 2. Overall, Lazy Leveling improves the complexity of update cost compared to leveling for all memory budgets while providing the same bounds on point lookups, space-amplification, and long range lookups, and providing a competitive bound on short range lookups.

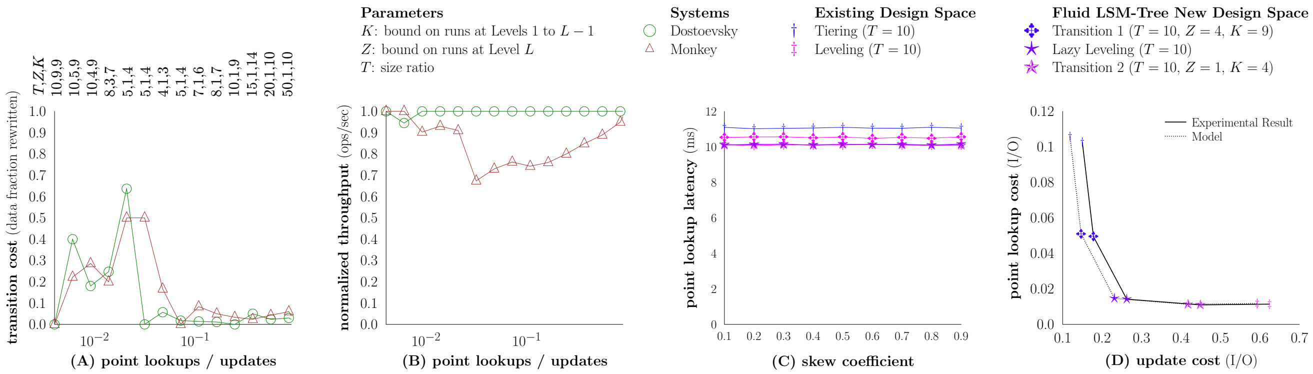


Figure 11: Dostoevsky dominates existing designs across workload transitions and lookup skews.

D RECURSIVE VS. PREEMPTIVE MERGING

A merge operation recurses to Level i if Levels 0 to $i-1$ are all at capacity. To avoid recursively copying data from smaller levels multiple times before they get merged into Level i, it is possible to preemptively merge all the runs that would eventually participate in a recursive merge by detecting from the onset that Levels 0 to $i-1$ are all at capacity. Recursive and preemptive merging are each best suited for different workloads. In particular, with high temporal skew for updates, many entries in smaller runs may get eliminated and so the merge may not trickle down to level i even if all smaller levels are full. In this case, it is desirable not to preemptively include Level i in the merge. On the other hand, for workloads with no temporal skew where every entry gets merged across all levels, preemptive merging is in fact the best strategy as it means the same entries do not get merged recursively multiple times but only once. In this work, our concern is managing space-time trade-offs in the worst-case, and so we use preemptive merging as it is the optimal strategy for the worst-case.

E RELATED TOPICS

Specialized Key-Value Stores. LSM-tree based Key-value stores have been proposed for specialized domains such as LogKB [18] for log processing, GeckoFTL[23] for Flash Translation Layers, and SlimDB [47] for semi-sorted data. Our work is general-purpose, we expect that our techniques would be able to push performance and space trade-offs for these specialized applications as well.

Main Memory Key-Value Stores. Key-value stores such as Redis [46] and Memcached [30] store data in main memory. This work focuses on key-value stores for secondary storage, and so it is orthogonal to memory based key-value stores. However, as there is a memory hierarchy for main memory systems including a hierarchy of caches, we expect that such a similar study would also benefit memory based systems.

Reducing Merge Costs. Recent designs reduce merge overheads by merging parts of runs with the least amount of key range overlap in the next level [12, 49, 51], or by opportunistically merging parts of runs that had recently been read and are therefore already cached in memory [43]. RocksDB [28] uses both techniques, and so our implementation on top of it takes advantage of them. Walnut [20] and WiscKey [39] store large values in a separate log to avoid merging them repeatedly at the cost of an extra level of indirection for lookups. This technique is compatible with Dostoevsky but it requires a slight modification to the cost model to only account for merging keys and for accessing the log during lookups.

Scheduling Merge Operations. To provide stable performance, all LSM-tree based key-value stores in the literature and industry de-amortize large merge operations by scheduling them evenly across time. Some designs schedule fractions of merge operations incrementally with respect to application updates [13, 35, 48], while other designs hide the cost of merge operations by performing them on dedicated servers [3]. LSM-trie [53] merges based on hashes of keys to restrict overlap. Most mainstream designs partition runs into SSTables and merge one SSTable at a time whenever a level reaches capacity [6, 19, 28, 31, 32, 34, 48, 52]. Our implementation on top of RocksDB takes advantage of this technique.

Unbounded Largest Level. Log-structured designs in the literature have have been proposed with fixed capacities for smaller levels and an unbounded capacity for the largest level, e.g., SILT [37] and MASM [9, 10]. Such designs can be thought of as consisting of a small log and a larger sorted array. When the log reaches a fixed size threshold of X entries, it gets merged into the sorted array. In such designs, update cost is proportional to $O(N/X)$, meaning it increases linearly with respect to the data size N. In contrast, Dostoevsky enforces fixed size ratio T among levels. Since the number of levels grows at a logarithmic rate with respect to N, and since every entry gets merged a constant number of times per level, update cost logarithmic in the data size: $O(\log_T(N))$. In this work, we restrict the scope of analysis to the latter class of designs due to their their logarithmic scalability property for updates, which is vital and thereby used in all key-value stores in industry [7, 19, 28, 34, 48, 52].

F EXPERIMENTAL CONTROL

Any experimentation on LSM-trees is subject to the problem that the cost of updates is not paid upfront but through subsequent merge operations. Therefore, running different experimental trials on the same dataset causes different trials to interfere with each other as performance for a given trial is influenced by updates issued by all the trials that preceded it. We addressed this problem differently for the fixed designs (i.e., the curves labeled Leveling, Tiering, Lazy Leveling, Transition 1, Transition 2, and Default RocksDB) and for the adaptive designs (i.e., the curved labeled Dostoevsky, Monkey, and Well-Tuned RocksDB) across the experiments in Figures 10 and 11. For the fixed designs, we created a baseline version from which we created a new clone for each trial. For the adaptive designs, we used cost models to find in advance the best tuning for the given workload, constructed an LSM-tree with this tuning, and then ran the experiment on it. Starting from a brand new LSM-tree for each trial eliminated interference among the trials thereby ensuring experimental control. Another positive feature of this approach is that it enforced experimental control by isolating the study of different designs from the study of how to transition among them (which is an issue we now explain separately in Appendix G).

G DYNAMIC RUNTIME ADAPTATION

Adaptation in Dostoevsky consists of three phases: Measurement, Computation, and Transition. In Section 4.3, we focused on the Computation Phase. We now fill in the details and evaluate the whole adaptation workflow.

Measurement Phase. The Measurement Phase in Dostoevsky is responsible for workload detection through time windows. The length of a window is a multiple of X buffer flushes during which we count the number of different operations in the workload and the average selectivities of range lookups. X is a parameter that controls a trade-off between the speed vs. the accuracy of finding the best tuning. In our implementation we set X to 16 by default, though it is possible to alleviate the trade-off between speed vs. accuracy through known smoothing techniques across the time windows [2, 40]. This phase involves no I/O, and so it does not contribute significantly to overall system overheads.

Computation Phase. At the end of the Measurement Phase, we run the Computation Phase to find the optimal tuning of Fluid LSM-tree given the workload statistics. We describe the algorithm and its runtime characteristics in Section 4.3. In practice, runtime takes up a fraction of a second and involves no I/O, and so this phase does not contribute significantly to system overheads.

Transition Phase. The Transition Phase physically adapts Fluid LSM-tree to the new target tuning found by the Computation Phase. Our transition strategy reconciles for the fact that the I/O cost of lookups is paid immediately whereas the I/O cost of updates is paid later through merge operations. It does this by adapting lazily or greedily depending on whether the new target tuning is more update-optimized or lookup-optimized relative to the current tuning, respectively. If the target tuning is more update-optimized, we adapt lazily by relaxing the merge policy so that the amortized cost of updates through subsequent merge operations is reduced. On the other hand, if the new target tuning is more lookup-optimized, we trigger merge operations greedily so that subsequent lookups become cheaper as quickly as possible.

Figure 11 (A) and (B) compare transition costs between Dostoevsky and Monkey across designs that are optimized for different workloads. The values on top of Figure 11 (A) for the parameters T, K, and Z reflect the optimal tunings used by Dostoevsky earlier for the experiment in Figure 10 (A) in Section 5, where we evaluated these designs independently of transition overheads. We now factor in transition overheads to show their impact on throughput. On the y-axis in Figure 10 (A), we measure the pure transition cost of Dostoevsky between adjacent designs in terms of the fraction of the overall data that was rewritten while transitioning. Despite some peaks, Dostoevsky merges 28% less than Monkey across all transitions (i.e., measured by the sizes of the areas underneath the curves). The reason is that Dostoevsky adapts across designs that are more similar to each other (i.e., Transition 2, Lazy Leveling, and Transition 1), whereas Monkey transitions across designs that are comparatively more different. In other words, the Dostoevsky design space is more dense and so optimal designs for similar workloads are structurally closer to each other and require less of a transition overhead. In Figure 10 (B), we show how transition overheads impact throughput. For each of these trials, we started with the structure represented by the previous point, transitioned to the new tuning, and continued to run the workload for a combined N update/lookup operations. We measure overall throughput including the time taken for transitions. Dostoevsky largely outperforms Monkey across the board because it offers more highly optimized designs for individual workloads as well as lower transition overheads across these designs. In fact, had we continued to run the workload for longer in each of these trials, the curves will have eventually converged to the ones in Figure 10 (A) as the transition costs become increasingly amortized during a stable workload.

Greedy transitioning for optimizing lookups is a single point in the space of possible policies. Future work directions include piggybacking transitions on regular merge operations to amortize transition costs, and scheduling transitions more cautiously to ensure that workload periodicity and volatility do not turn transitions into pure overhead. This topic deserves a much larger study, whereas the current adaptive policies are a starting point to demonstrate that the design space is worth navigating.

H ROBUST POINT LOOKUP PERFORMANCE ACROSS SKEWS

In Figure 11 (C), we verify that our experimental results generalize across a wide range of lookup skews. In the experiment, a fraction of c of the lookups target the least recently inserted fraction of $1 - c$ of the entries. The horizontal lines show that point lookup cost is insensitive to skew across all designs. The intuition is that most of the data is at the largest level and so most lookups that target recently inserted entries still reach the largest level most of the time. As a result, the models that Dostoevsky uses to transition across designs are robust across workload skews.

I MODEL VALIDATION

In Figure 11 (D), we validate our I/O cost models against the experimental results. We created this figure by comparing the number of I/Os per operation that we observe in practice vs. the number of I/Os predicted by our cost models in Section 4.2. Every point along the line labeled Experimental Result is the average taken from thousands of updates and lookups. As shown, the experimental results have a slightly higher update cost than predicted by our model. The reason is that our model does not take account of I/Os issued by writing to RocksDB's log. Moreover, the experimental results have a slightly lower zero-result point lookup cost. The reason is that runs in RocksDB are partitioned and merged as units of SSTables. When an SSTable is merged into the next level, it leaves a hole in the key-space at its origin level [51]. RocksDB is aware of such holes by maintaining the first and last key of every SSTable in main memory, and so it eliminates the chance of a false positive by avoiding access to the level altogether if the target key targets a key range for which there is a hole. Despite these sources of error, our model provides a good fit for the experimental results.