The VLDB Journal
https://doi.org/10.1007/s00778-020-00638-1

REGULAR PAPER

Comparison and evaluation of state-of-the-art LSM merge policies

Qizhong Mao[1] · Steven Jacobs[1] · Waleed Amjad[1] · Vagelis Hristidis[1] · Vassilis J. Tsotras[1] · Neal E. Young[1]

Received: 29 March 2020 / Revised: 30 September 2020 / Accepted: 22 December 2020

Abstract
Modern NoSQL database systems use log-structured merge (LSM) storage architectures to support high write throughput. LSM architectures aggregate writes in a mutable *MemTable* (stored in memory), which is regularly flushed to disk, creating a new immutable file called an *SSTable*. Some of the SSTables are chosen to be periodically *merged*—replaced with a single SSTable containing their union. A *merge policy* (a.k.a. compaction policy) specifies when to do merges and which SSTables to combine. A *bounded depth* merge policy is one that guarantees that the number of SSTables never exceeds a given parameter k, typically in the range 3–10. Bounded depth policies are useful in applications where low read latency is crucial, but they and their underlying combinatorics are not yet well understood. This paper compares several bounded depth policies, including representative policies from industrial NoSQL databases and two new ones based on recent theoretical modeling, as well as the standard Tiered policy and Leveled policy. The results validate the proposed theoretical model and show that, compared to the existing policies, the newly proposed policies can have substantially lower write amplification with comparable read amplification.

Keywords NoSQL database · LSM · Merge policy · Compaction

1 Introduction

Many modern NoSQL systems [8,9] use log-structured merge (LSM) architectures [36] to achieve high write throughput. To insert a new record, a WRITE operation simply inserts the record into the memory-resident *MemTable* [9] (also called the in-memory component). UPDATE operations are implemented lazily, requiring only a single WRITE to the MemTable. DELETE operations are implemented similarly, by writing an *anti-matter* record for the key to the MemTable. Thus, each WRITE, UPDATE, or DELETE operation avoids any immediate disk access. When the MemTable reaches its allocated capacity (or for other reasons), it is flushed to disk, creating an immutable disk file called a component, or, usually, an *SSTable* (Sorted Strings Table [9]). This process continues, creating many SSTables over time.

Each READ operation searches the MemTable and SSTables to find the most recent value written for the given key. With a compact index stored in memory for each SSTable, checking whether a given SSTable contains a given key typically takes just one disk access [22, §2.5]. (For small SSTables, this access can sometimes be avoided by storing a Bloom filter for the SSTable in memory [14].) Hence, the time per READ grows with the number of SSTables. To control READ costs, the system periodically *merges* SSTables to reduce their number and to prune updated and anti-matter records. Each merge replaces some subset of the SSTables by a single new SSTable that holds their union. The merge batch-writes these items to the new SSTable on disk. The *write amplification* is the number of bytes written by all merges, divided by the number of bytes inserted by WRITE operations.

A *merge policy* (also known as a *compaction* policy) determines how merges are done. The policy must efficiently trade off total write amplification for total read cost

✉ Qizhong Mao
qmao002@ucr.edu

Steven Jacobs
sjaco002@ucr.edu

Waleed Amjad
wamja001@ucr.edu

Vagelis Hristidis
evangelo@ucr.edu

Vassilis J. Tsotras
vtsotras@ucr.edu

Neal E. Young
neal.young@ucr.edu

1 Department of Computer Science and Engineering, University of California, Riverside, CA, USA

(which increases with the average number of SSTables being checked per READ operation, known as *read amplification*). This paper focuses on what we call *bounded depth* policies—those that guarantee a bounded number of disk accesses for each READ operation by ensuring that, at any given time, the SSTable count (the number of existing SSTables) never exceeds a given parameter k, typically 3–10, such that the read amplification is at most k. Maintaining bounded depth is important in applications that require low read latency, but bounded depth policies are not yet well understood.

A recent theoretical work by Mathieu et al. [35] (including one of the current authors) formally defines a broad class of so-called *stack-based policies*. (See Sect. 3 for the definition.) This class includes policies of many popular NoSQL systems, including Bigtable [9], HBase [19,27,37], Accumulo [26,37], Cassandra [29], Hypertable [25], and AsterixDB [2]. In contrast, *leveled* policies (used by LevelDB and its spin-offs [21]) split SSTables by key space to avoid monolithic merges, so they do not fit the stack-based model. Note that all current leveled implementations yield unbounded depth; hence, they are not considered here.

Mathieu et al. [35] also propose theoretical metrics for policy evaluation and, as a proof of concept, propose new policies that, among stack-based policies, are optimal according to those metrics. Two such policies, MINLATENCY and BINOMIAL (defined in Sect. 2) are bounded depth policies which were designed to have minimum *worst-case write amplification* (subject to the depth constraint) among all stack-based policies. Mathieu et al. [35] observe that, according to the theoretical model, on some inputs *existing policies are far from optimal*, so, on some common workloads, compared to existing policies, MINLATENCY and BINOMIAL can have lower write amplification.

Here, we empirically compare MINLATENCY and BINOMIAL to three representative bounded depth merge policies from state-of-the-art NoSQL databases: a policy from AsterixDB [6], EXPLORING (the default policy for HBase [5]), and the default policy from Bigtable (as described by Mathieu et al. [35], which includes authors from Google), as well as the standard TIERED policy (the default policy for Cassandra [4]) and LEVELED policy (the default policy for LevelDB [21]). Section 2 defines these policies. We implement the policies under consideration on a common platform—Apache AsterixDB [2,6]—and evaluate them on inputs from the Yahoo! Cloud Serving Benchmark (YCSB) [11,44]. This is the first implementation and evaluation of the policies proposed by Mathieu et al. [35] on a real NoSQL system. The empirical results validate the theoretical model. MINLATENCY and BINOMIAL achieve write amplification close to the theoretical minimum, thereby outperforming the other policies by orders of magnitude on some realistic workloads. (See Sect. 4.)

Having a realistic theoretical model facilitates merge policy design both via theoretical analysis (as for MINLATENCY and BINOMIAL), and because it enables rapid but faithful simulation of experiments. NoSQL systems are designed to run for months, incorporating hundreds of terabytes. Experiments can take weeks, even with appropriate adaptations. In contrast, the model allows some experiments to be faithfully simulated in minutes. (See Sect. 5.)

In summary, this work makes the following contributions:

1. The implementation of several existing merge policies, including the popular TIERED and LEVELED, and two recently proposed merge policies, on a common, open-source platform, specifically Apache AsterixDB.
2. An experimental evaluation on write, read, and transient space amplification using the Yahoo! Cloud Serving Benchmark (YCSB) benchmark, confirming that the recently proposed policies can significantly outperform the state-of-the-art policies on some common workloads, such as append-only and update-heavy workloads.
3. A study on how insertion order affects the write amplification of merge policies, especially for LEVELED.
4. We have shown that BINOMIAL and MINLATENCY outperform the popular TIERED and LEVELED policies with a better trade-off between write amplification and average read amplification.
5. An empirical validation of a realistic cost model, which facilitates the design of merge policies via theoretical analysis and rapid simulation.

2 Policies studied

Bigtable (Google) The default for the Bigtable platform is as follows [35]. *When the MemTable is flushed, if there are fewer than k SSTables, add a single new SSTable holding the MemTable contents. Otherwise, merge the MemTable with the i most recently created SSTables, where i is the minimum such that, afterwards, the size of each SSTable exceeds the sum of the sizes of all newer SSTables.*[1] Roughly speaking, this tries to ensure that each SSTable is at most half the size of the next older SSTable. We denote this policy BIGTABLE.

Exploring (Apache HBase) EXPLORING is the default for HBase [5]. In addition to $\boldsymbol{k}$, it has configurable parameters $\boldsymbol{\lambda}$ (default 1.2), $\boldsymbol{C}$ (default 3), and $\boldsymbol{D}$ (default 10). When the MemTable (Memstore in HBase) is flushed, the policy orders the SSTables (HFiles in HBase) by time of creation, considers various contiguous subsequences of them, and merges one that is in some sense most cost-effective. Specifically: *Temporarily add the MemTable as its own (newest) SSTable and then consider every contiguous subsequence s such that*

[1] The implementation of this and other policies may temporarily create an SSTable holding the MemTable contents and then merge that SSTable with the other SSTables.

- *s has at least C and at most D SSTables, and*
- *in s, the size of the largest SSTable is at most λ times the sum of the sizes of the other SSTables.*

In the case that there is at least one such subsequence s, merge either the longest (if there are at most k SSTables) or the one with minimum average SSTable size (otherwise). In the remaining case, and only if there are more than k SSTables, merge a contiguous subsequence of C SSTables having minimum total size.

Constant (AsterixDB before version 0.9.4) CONSTANT is as follows. *When the MemTable is flushed, if there are fewer than k SSTables, add a single new SSTable holding the MemTable contents. Otherwise, merge the MemTable and* all *k SSTables into one.*

Tiered and Leveled TIERED policy is the default for Cassandra. LEVELED is the default for LevelDB [21]. In theory, both policies have one core configurable parameter, the size ratio ***B***. In practice, TIERED may need multiple parameters (3 in Cassandra) to determine SSTables of similar sizes, and LEVELED also has an extra parameter that control the number of SSTables in level 0 as an on disk buffer. The total SSTable size in one tier or level is ***B*** times larger than the previous tier or level. The differences are:

- In TIERED, every tier must have at most ***B*** SSTables and each SSTable is ***B*** larger than the SSTable size in the previous tier. In LEVELED, all SSTables are of the same size and the number of SSTables in one level is ***B*** more than the previous level.
- Any two SSTables can have overlapping key space in TIERED, while all SSTables must not have overlapping key space in the same level in LEVELED
- TIERED only allows merging consecutive SSTables, while in LEVELED, one SSTable is picked to be merged with all SSTables in the next level that have overlapping key ranges with the picked SSTable (if any). These SSTables do not have to be consecutive.
- In TIERED, every merge involves at least two SSTables. In LEVELED, only one SSTable can be merged if there is no overlapping SSTable in the next level.
- Only one SSTable is created in a merge in TIERED. In LEVELED, the number of SSTables created in a merge is typically the same as the number of SSTables being merged.
- In TIERED, the new SSTable size is typically the same as the total size of the SSTables being merged. In LEVELED, all input and output SSTables have the same size.

Next are the definitions of the MINLATENCY and BINOMIAL policies which were proposed by Mathieu et al [35]. First, define a utility function B, as follows. Consider any binary search tree T with some nodes $\{1, 2, \ldots, n\}$ in search tree order (each node is larger than those in its left subtree, and smaller than those in its right subtree). Given a node t in T, define its *stack (merge) depth* to be the number of ancestors smaller (larger) than t. (Hence, the depth of t in T equals its stack depth plus its merge depth.)

Fix any two positive integers k and m, and let $n = \binom{m+k}{k} - 1$. Let $\tau^*(m,k)$ be the unique n-node binary search tree on nodes $\{1, 2, \ldots, n\}$ that has maximum stack depth $k-1$ and maximum write depth $m-1$. For $t \in \{1, 2, \ldots, n\}$, define $B(m,k,t)$ to be the stack depth of node t in T.

Compute the function $B(m,k,t)$ via the following recurrence. Define $B(m,k,0)$ to be zero, and for $t > 0$ use

$$B(m,k,t) = \begin{cases} B(m-1,k,t) & \text{if } t < \binom{m+k-1}{k}, \\ 1 + B\left(m, k-1, t - \binom{m+k-1}{k}\right) & \text{if } t \geq \binom{m+k-1}{k}. \end{cases}$$

The policies are defined as follows.

MinLatency For each $t = 1, 2, \ldots, n$, in response to the tth flush, the action of the policy is determined by t, as follows:

Let $m' = \min\{m : \binom{m+k}{m} > t\}$ *and* $i = B(m',k,t)$. *Order the SSTables by time of creation, and merge the i-th oldest SSTable with all newer SSTables and the flushed MemTable (leaving i SSTables).*

Binomial For each $t = 1, 2, \ldots, n$, in response to the tth flush, the action of the policy is determined by t, as follows:

Let $T_k(m) = \sum_{i=1}^{m} \binom{i+\min(i,k)-1}{i}$ *and* $m' = \min\{m : T_k(m) \geq t\}$.

Let $i = 1 + B(m', \min(m',k) - 1, t - T_k(m'-1) - 1)$. *Order the SSTables by time of creation, and merge the i-th oldest SSTable with all newer SSTables and the flushed MemTable (leaving i SSTables).*

As described in Sect. 3, these policies are designed carefully to have the minimum possible *worst-case write amplification* among all policies in the aforementioned class of stack-based policies.

BIGTABLE, CONSTANT, and (although it is not obvious from its specification) MINLATENCY are *lazy*—whenever the MemTable is flushed, if there are fewer than k SSTables, the policy leaves those SSTables unchanged and creates a new SSTable that holds just the flushed MemTable's contents. For this reason, these policies tend to keep the number of SSTables close to k. In contrast, for moderate-length runs (4^k or fewer flushes, as discussed later), EXPLORING and BINOMIAL often merge multiple SSTables even when fewer than k SSTables are already present, so may keep the average number of SSTables well below k, potentially allowing faster READ operations.

Examples of all these seven merge policies for the first 16 flushes are shown in Fig. 1. For the six stack-based policies (Fig. 1a–f), a new SSTable is added to the top of the stack in every flush. Several SSTables are merged into one SSTable. For example in Fig. 1a, before the 12th flush, there are 2

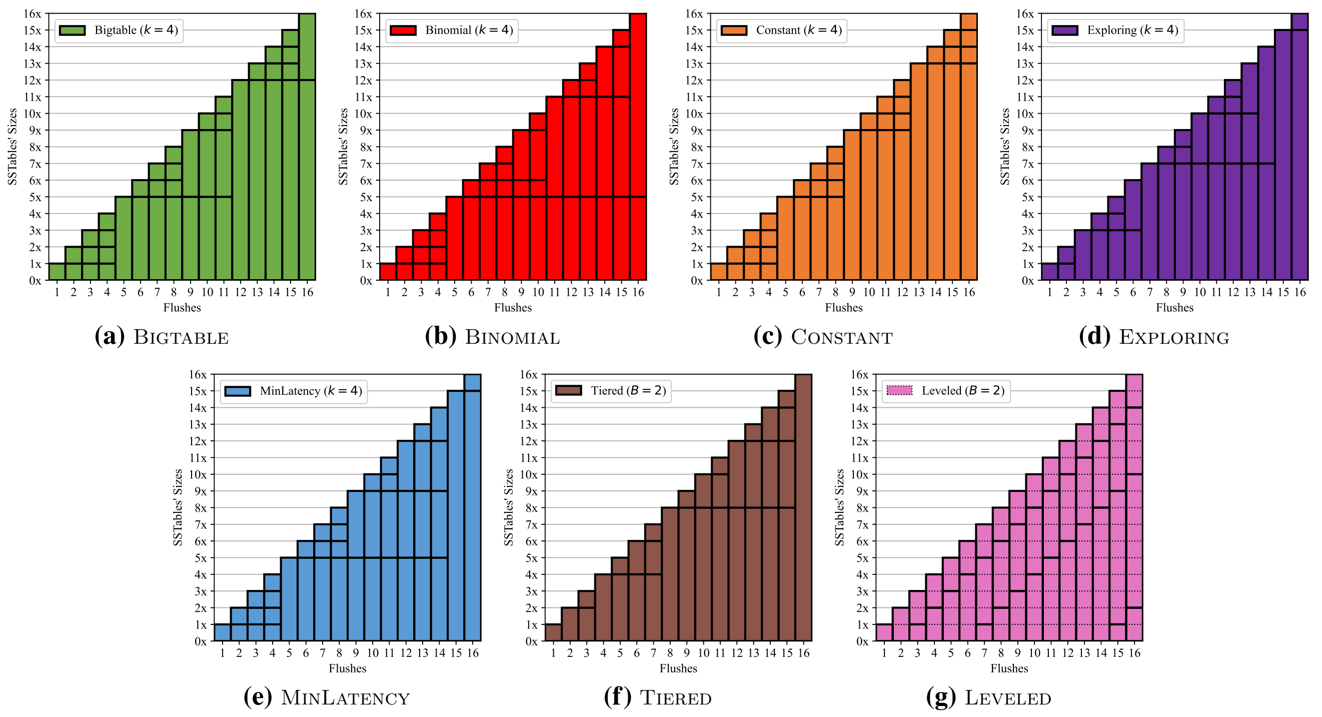


Fig. 1 Examples of SSTables states after 16 flushes for the 7 merge policies. SSTables are represented by rectangles (solid for the 6 stack-based policies, dotted for LEVELED). Older SSTables are lower, newer SSTables are higher. Levels are represented by solid rectangles in the last plot. SSTables' sizes are with respect to the flush size

SSTables of size 2x and 2 SSTables of size 1x. After the 12th flush, all the 5 SSTables are merged into one big SSTable of size 12x. The number of SSTables of BIGTABLE, BINOMIAL, CONSTANT, EXPLORING, and MINLATENCY never exceeds $k = 4$. BINOMIAL and MINLATENCY choose different SSTables to merge starting from the 9th flush, based on their own computations (Fig. 1b, e). For TIERED, a merge is triggered every $B = 2$ flushes and multiple merges are triggered at the 4th, 8th, 12th, and 16th flush (Fig. 1f). For LEVELED, multiple merges may be triggered at every flush starting from the 3rd flush, while only one merge is triggered at the 2nd flush (Fig. 1g). For example, before the 12th flush, there are 2 SSTables in level 1 (top rectangle), 4 rectangles in level 2, and 5 SSTables in level 3 (bottom rectangle). After the 12th flush, a new SSTable is added to level 1, triggering a merge which selects an SSTable in level 1 and merges it to level 2. Then, level 2 has 5 SSTables, and another merge is triggered which selects an SSTable in level 2 and merges it to level 3. Eventually, there are still 2 SSTables in level 1 and 4 SSTables in level 2, but 6 SSTables in level 3.

3 Design of MinLatency and Binomial

This section reviews definition of the class of so-called *stack-based* merge policies in [35], the *worst-case write amplification* metric, and how MINLATENCY and BINOMIAL are designed to minimize that metric among all policies in that class.

3.1 Bounded depth stack-based merge policies

Informally, a *stack-based policy* must maintain a set of SSTables over time. The set is initially empty. At each time $t = 1, 2, \ldots, n$, the MemTable is flushed, having current size in bytes equal to a given integer $\ell_t \geq 0$. In response, the merge policy must choose some of its current SSTables and then replace those chosen SSTables by a single SSTable holding their contents and the MemTable contents. As a special case, the policy may create a new SSTable from the MemTable contents alone. (The policy may replace additional sets of SSTables by their respective unions, but the policies studied here do not.)

Each newly created SSTable is written to the disk, batch-writing a number of bytes equal to its size, which *by assumption* is the sum of the sizes of the SSTables it replaces, plus ℓ_t if the merge includes the flushed MemTable. (This ignores UPDATEs and DELETEs, but see the discussion below.)

A *bounded depth* policy (in the context of a parameter k) must keep the SSTable count at k or below. Subject to that constraint, its goal is to minimize the *write amplification*, which is defined to be the total number of bytes written in creating SSTables, divided by $\sum_{t=1}^{n} \ell_t$, the sum of the sizes of the n MemTable flushes. (Write amplification is a standard measure in LSM systems [16,30,31,40].) TIERED is stack-based but not bounded depth, while LEVELED is neither stack-based nor bounded depth.

For intuition, consider the example $k = 2$ and $\ell_t = 1$ uniformly for $t \in \{1, 2, \ldots, n\}$. The optimal write amplification is $\Theta(\sqrt{n})$.

Next is the precise formal definition, as illustrated in Fig. 2a:

Problem 1 (k-Stack-Based LSM Merge) A problem instance is an $\ell \in \mathbb{R}_+^n$. For each $t \in \{1, \ldots, n\}$, say *flush* t has *(flush) size* ℓ_t. A solution is a sequence $\sigma = \{\sigma_1, \ldots, \sigma_n\}$, called a *schedule*, where each σ_t is a partition of $\{1, 2, \ldots, t\}$ into at most k parts, each called an *SSTable*, such that σ_t is refined by[2] $\sigma_{t-1} \cup \{\{t\}\}$ (if $t \geq 2$). The *size* of any SSTable F is defined to be $\ell(F) = \sum_{t \in F} \ell_t$—the sum of the sizes of the flushes that comprise F. The goal is to minimize σ's *write amplification*, defined as $W(\sigma) = \sum_{t=1}^{n} \delta(\sigma_t, \sigma_{t-1}) / \sum_{t=1}^{n} \ell_t$, where $\delta(\sigma_t, \sigma_{t-1}) = \sum_{F \in \sigma_t \setminus \sigma_{t-1}} \ell(F)$ is the sum of the sizes of the new SSTables created during the merge at time t.

Formally, a *(bounded depth) stack-based merge policy* is a function P mapping each problem instance $\ell \in \mathbb{R}_+^n$ to a solution σ. In practice, the policy must be *online*, meaning that its choice of merge at time t depends only on the flush sizes $\ell_1, \ell_2, \ldots, \ell_t$ seen so far. Because future flush sizes are unknown, no online policy P can achieve minimum possible write amplification for *every* input ℓ. Among possible metrics for analyzing such a policy P, the focus here is on *worst-case* write amplification: the maximum, over all inputs $\ell \in \mathbb{R}_+^n$ of size n, of the write amplification that P yields on the input. Formally, this is the function $n \mapsto \max\{W(P(\ell)) : \ell \in \mathbb{R}_+^n\}$.
Updates and Deletes The formal definitions above ignore the effects of key UPDATEs and DELETEs. While it would not be hard to extend the definition to model them, for designing policies that minimize worst-case write amplification, this is unnecessary: These operations only *decrease* the write amplification for a given input and schedule, so any online policy in the restricted model above can easily be modified to achieve the same worst-case write amplification, even in the presence of UPDATEs and DELETEs.
Additional terminology Recall that a policy is *stable* if, for every input, it maintains the following invariant at all times among the current SSTables: *the* WRITE *times of all items in any given SSTable precede those of all items in every newer SSTable.* (Formally, every SSTable created is of the form $\{i, i+1, \ldots, j\}$ for some i, j.) As discussed previously, this can speed up READs. We note without proof that any unstable solution can be made stable while at most doubling the write amplification. Likewise, each uniform input has an optimal stable solution. All policies tested here are stable.

[2] Each part in σ_t is the union of some parts in $\sigma_{t-1} \cup \{\{t\}\}$.

A policy is *eager* if, for every input ℓ, for every time t, the policy creates just one new SSTable (necessarily including the MemTable flushed at time t). Every input has an optimal eager solution, and all bounded depth policies tested here except for EXPLORING are eager.

An online policy is *static* if each σ_t is determined *solely by* k *and* t. In a static policy, the merge at each time t is predetermined—for example, for $t = 1$, merge just the flushed MemTable; for $t = 2$, merge the MemTable with the top SSTable, and so on—independent of the flush sizes $\ell_1, \ell_2, \ldots$ The MINLATENCY and BINOMIAL policies are static. Static policies ignore the flush sizes, so it may seem counterintuitive that static policies can achieve optimum worst-case write amplification.

3.2 MinLatency and Binomial

Among bounded depth stack-based policies, MINLATENCY and BINOMIAL, by design, have the minimum possible worst-case write amplification. Their design is based on the following relationship between schedules and binary search trees.

Fix any k-Stack-based LSM Merge instance $\ell = (\ell_1, \ldots, \ell_n)$. Consider any eager, stable schedule σ for ℓ. (So σ creates just one new SSTable at each time t.) Define the (rooted) *merge forest* $\mathcal{F}$ for σ as follows: For $t = 1, 2, \ldots, n$, represent the new SSTable F_t that σ creates at time t by a new node t in $\mathcal{F}$, and, for each SSTable F_s (if any) that is merged in creating F_t, make node t the parent of the node s that represents F_s.

Next, create the *binary search tree* T for σ from $\mathcal{F}$ as follows. Order the roots of $\mathcal{F}$ in decreasing order (decreasing creation time t). For each node in $\mathcal{F}$, order its children likewise. Then, let $T = T(\sigma)$ be the standard left-child, right-sibling binary tree representation of $\mathcal{F}$. That is, T and $\mathcal{F}$ have the same vertex set $\{1, 2, \ldots, n\}$, and, for each node t in T, the left child of t in T is the first (oldest) child of t in $\mathcal{F}$ (if any), while the right child of t in T is the right (next oldest) sibling of t in $\mathcal{F}$ (if any; here we consider the roots to be siblings). It turns out that (because σ is stable) the nodes of T must be in search-tree order. (Each node is larger than those in its left subtree and smaller than those in its right subtree.) Figure 2a, c shows an example.

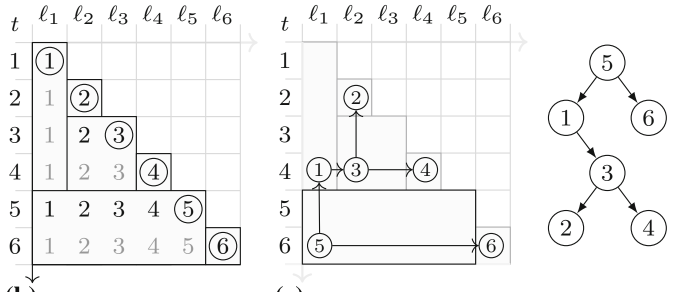
What about the depth constraint on σ, and its write amplification? Recall that the *stack (merge) depth* of a node t is the number of ancestors that are smaller (larger) than t. While the details are out of scope here, the following holds:

For any eager, stable schedule σ:

1. σ obeys the depth constraint if and only if every node in $T(\sigma)$ has stack depth at most $k - 1$,
2. the write amplification incurred by σ on ℓ equals

t	SSTables at time t	bytes written	SSTable count
1	$\sigma_1 = \{\mathbf{1}\}$	ℓ_1	1
2	$\sigma_2 = \{1\}, \{\mathbf{2}\}$	ℓ_2	2
3	$\sigma_3 = \{1\}, \{\mathbf{2}, \mathbf{3}\}$	$\ell_2 + \ell_3$	2
4	$\sigma_4 = \{1\}, \{2, 3\}, \{\mathbf{4}\}$	ℓ_4	3
5	$\sigma_5 = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}$	$\ell_1 + \ell_2 + \ell_3 + \ell_4 + \ell_5$	1
6	$\sigma_6 = \{1, 2, 3, 4, 5\}, \{\mathbf{6}\}$	ℓ_6	2

(a)



(b) **(c)**

Fig. 2 **a** An eager, stable schedule σ ($n = 6, k = 3$). **b** A graphical representation of σ. Each shaded rectangle is an SSTable (over time). Row t is the stack at time t. **c** The binary-search-tree representation of σ

$$\frac{\sum_{t=1}^{n}(\mathsf{mergedepth}(t, T(\sigma)) + 1)\ell_t}{\sum_{t=1}^{n}\ell_t}$$
$$\leq 1 + \max_{t=1}^{n} \mathsf{mergedepth}(t, T(\sigma)).$$

The mapping $\sigma \rightarrow T(\sigma)$ is invertible. Hence, *any binary search tree t with nodes* $\{1, 2, \ldots, n\}$, *maximum stack depth* $k - 1$, *and maximum merge depth* $m - 1$ *yields a bounded depth schedule* σ *(such that* $T(\sigma) = t$*), having write amplification at most m on any input* $\ell \in \mathbb{R}_+^n$.

Rationale for MinLatency MinLatency uses this observation to produce its schedule [35]. First consider the case that $n = \binom{m+k}{k} - 1$ for some integer m. Among the binary search trees on nodes $\{1, 2, \ldots, n\}$, there is a unique tree with maximum stack depth $k - 1$ and maximum merge depth $m - 1$. Let $\tau^*(m, k)$ denote this tree, and let $\sigma^*(m, k)$ denote the corresponding schedule.

MinLatency is designed to output $\sigma^*(m, k)$ for any input of size n. Since $\tau^*(m, k)$ has maximum merge depth $m - 1$, as discussed above, $\sigma^*(m, k)$ has write amplification at most m, which by calculation is

$$(1 + O(1/k))\, k\, n^{1/k}/c_k, \tag{1}$$

where $c_k = (k + 1)/(k!)^{1/k} \in [2, e]$. This bound extends to arbitrary n, so MinLatency's worst-case write amplification is at most (1).

This is optimal, in the following sense: For every $\epsilon > 0$ and large n, no stack-based policy achieves worst-case write amplification less than $(1 - \epsilon)k\, n^{1/k}/c_k$. This is shown by using the bijection described above to bound the minimum possible write amplification for *uniform* inputs.

Binomial and the small-n and large-n regimes As mentioned previously, due to the fact that MinLatency and Bigtable are lazy, they produce schedules whose average SSTable count is close to k. When n is large, any policy with near-optimal write amplification must do this. Specifically, in what we call the *large-n regime*—after the number of flushes exceeds 4^k or so—any schedule with near-optimal write amplification (e.g., for uniform ℓ) *must* have average SSTable count near k. In this regime, Binomial behaves similarly to MinLatency. Consequently, in this regime, Binomial still has minimum worst-case write amplification.

However, in what we call the *small-n regime*—until the number of flushes n reaches 4^k—it is possible to achieve near-optimal write amplification while keeping the average SSTable count somewhat smaller. Binomial is designed to do this [35]. In the small-n regime, it produces the schedule σ for the tree $\tau^*(m, m)$, for which the maximum stack depth and maximum merge depth are both $m \approx \log_2(n)/2$, so Binomial's average SSTable count and write amplification are about $\log_2(n)/2$, which is at most k (in this regime) and can be less. Consequently, in the small-n regime, Binomial can opportunistically achieve average SSTable count well below k. In this way, it compares well to Exploring, and it behaves well even with unbounded depth ($k = \infty$).

4 Experimental evaluation

4.1 Test platform: AsterixDB

Apache AsterixDB [2,6] is a full-function, open-source big data management system (BDMS), which has a shared-nothing architecture, with each node in an AsterixDB cluster managing one or more storage and index partitions for its datasets based on LSM storage. Each node uses its memory for a mix of storing MemTables of active datasets, buffering of file pages as they are accessed, and other memory-intensive operations. AsterixDB represents each SSTable as a B^+-tree, where the number of keys at each internal node is roughly the configured page size divided by the key size. (Internal nodes store keys but not values.) Secondary indexing is also available using B^+-trees, R-trees, and/or inverted indexes [3]. As secondary indexing is out of the scope of this paper, our experiments involve only primary indexes.

AsterixDB provides *data feeds* for rapid ingestion of data [23]. A *feed adapter* handles establishing the connection with a data source, as well as receiving, parsing, and translating data from the data source into ADM objects [2] to be stored in AsterixDB. Several built-in feed adapters available for retrieving data from network sockets, local file system, or from applications like Twitter and RSS.

4.2 Experimental setup

The experiments were performed on a machine with an Intel i3–4330 CPU running CentOS 7 with 8 GB of RAM and two mirrored (RAID 1) 1 TB hard drives. AsterixDB was configured to use 1 node controller, so all records are stored on the same disk location. The relatively small RAM size of 8 GB limits caching, to better simulate large workloads. The MemTable capacity was configured at 4 MB. The small MemTable capacity increases the flush rate to better simulate longer runs.

The workload was generated using the Yahoo! Cloud Serving Benchmark (YCSB) [11,44], with default parameters used in load phase. The full workload consists of 80,000,000 WRITEs, each writing one record with a primary key of 5 to 23 bytes plus 10 attributes of 100 bytes each, giving 11 attributes of about 1 KB total size. Each primary key is a string with a 4-byte prefix and a long integer (as a string). Insert order was set to the default *hashed*.

To achieve high ingestion rate, we implemented a YCSB database interface layer for AsterixDB using the "socket _adapter" data feed (which retrieves data from a network socket) with an upsert data model, so that records are written without a duplicate key check to achieve a much higher throughput. Upsert in AsterixDB and Cassandra is the equivalent of standard insert in other NoSQL systems, where, if an inserted record conflicts in the primary key with an existing record, it overwrites it.

The MemTable flushes were triggered by AsterixDB when the MemTable was near capacity, so the input ℓ generated by the workload was nearly uniform, with each flush size ℓ_t about 4 MB. This represents about 3300 records per flush, so the input size n—the total number of flushes in the run—was just over 24,000.

For each of the five bounded depth stack-based policies tested, and for each $k \in \{3, 4, 5, 6, 7, 8, 10\}$, we executed a single run testing that policy and configured with that depth (SSTable count) limit k. For TIERED and LEVELED policies, we executed the same runs with size ratio $B \in \{4, 8, 16, 32\}$, the number of SSTables in level 0 was set to 2 in LEVELED policy. LEVELED also used a strategy that picks the SSTable which overlaps with the minimum number of SSTables in the next level for merges in order to reduce the write amplification. All other policy parameters were set to their default values. (See Sect. 2.) Each of the 43 runs started from an empty instance and then inserted all records of the workload into the database, generating just over 24,000 flushes for the merge policy.

For some smaller k values, some of the bounded depth policies had significantly large write amplification and so did not finish the run. BINOMIAL and MINLATENCY finished in about 16 hours, but BIGTABLE and EXPLORING ingested less than 40% of the records after two days, so were terminated early. Similarly, CONSTANT was terminated early in *all* of its runs.

As our focus is on write amplification, which is not affected by READs, the workload contains no READs. (But see Sect. 4.3.2.)

The data for all 43 runs are tabulated in "Appendix A."

4.3 Policy comparison

4.3.1 Write amplification

At any given time t during a run, define the *write amplification* (so far) to be the total number of bytes written to create SSTables so far divided by the number of bytes flushed so far ($\sum_{s=1}^{t} \ell_s$). This section illustrates how write amplification grows over time during the runs for the various policies. The 5 bounded depth stack-based policies all share a common parameter k, which is the maximum number of SSTables. On the other hand, TIERED and LEVELED both share a different parameter B, which is the size ratio between tiers or levels. Because these two parameters carry different meanings, it is not meaningful to compare the write amplification of these 7 policies directly with the same value of k and B. Thus, in this subsection, we compare and evaluate them into 2 groups: One group containing the 5 bounded depth stack-based policies are compared for the same value of k, while the other group of TIERED and LEVELED are compared for the same value of B.

Bounded depth stack-based policies

We focus on the runs with $k \in \{5, 6, 7\}$, which are particularly informative. The runs for each k are shown in Fig. 3a–c, each showing how the write amplification grows over the course of all $n \approx 24{,}000$ flushes. Because workloads with at most a few thousand flushes are likely to be important in practice, Fig. 3d–f repeats the plots, zooming in to focus on just the first 2000 flushes ($n = 2000$).

In interpreting the plots, note that the caption of each subfigure shows the threshold 4^k. The small-n regime lasts until the number of flushes passes this threshold, whence the large-n regime begins. Note that (depending on n and k) some runs lie entirely within the small-n regime ($n \leq 4^k$), some show the transition, and in the rest (with $n \gg 4^k$) the small-n regime is too small to be seen clearly. In all cases, the results depend on the regime as follows. During the small-n

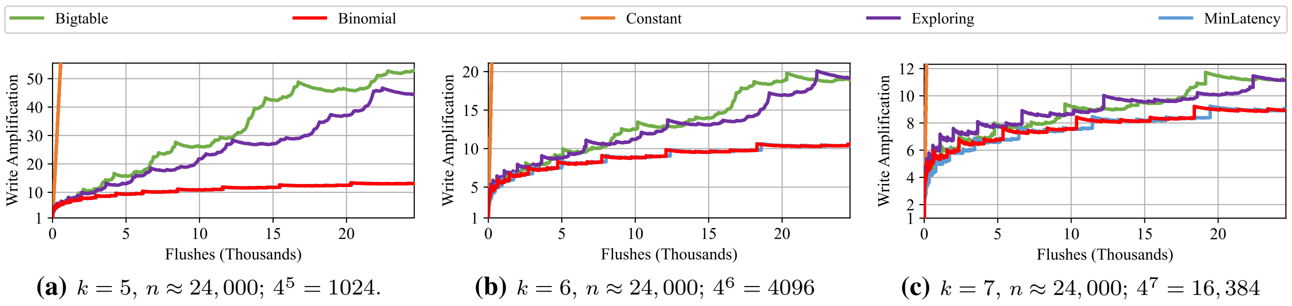
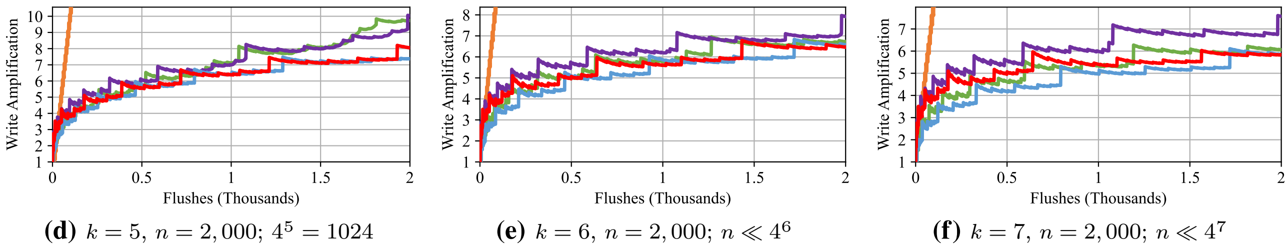



Fig. 3 Write amplification versus number of flushes over time for runs with $k \in \{5, 6, 7\}$. The top row shows $n \approx 24{,}000$; the bottom shows $n = 2000$. The transition from the small-n regime to the large-n regime (if present) occurs at 4^k flushes. Complete data are in "Appendix A"

regime, MinLatency has smallest write amplification, with Binomial, Bigtable, and then Exploring close behind. As the large-n regime begins, MinLatency and Binomial become indistinguishable. Their write amplification at time t grows sub-linearly (proportionally to $t^{1/k}$), while those of Bigtable and Exploring grow linearly (proportionally to t). Although we do not have enough data for Constant, its write amplification is $O(t/k)$ as it merges all SSTables in every k flushes. These results are consistent with the analytical predictions from the theoretical model [35].

Tiered policy and Leveled policy

The runs for Tiered and Leveled are shown in Fig. 4a for $n \approx 24{,}000$ flushes with $B \in \{4, 8, 16, 32\}$. Tiered achieved lowest write amplification than any other policy tested, while Leveled has significantly higher write amplification than all the other policies except for Constant. The write amplification is $O(\log t)$ and $O(B \log t)$ for Tiered and Leveled, respectively. From the figure, it is observable that smaller size ratio leads to higher write amplification in Tiered but lower write amplification in Leveled, which verifies the theoretical numbers. Runs with 2000 flushes are shown in Fig. 4b which shows the same results. Unlike the bounded depth policies, the small-n regime does not apply to Tiered and Leveled.

4.3.2 Read amplification

Read amplification is the number of disk I/Os per read operation. In this paper, we focus on point query only. In practice, accessing one SSTable only costs one disk I/O,

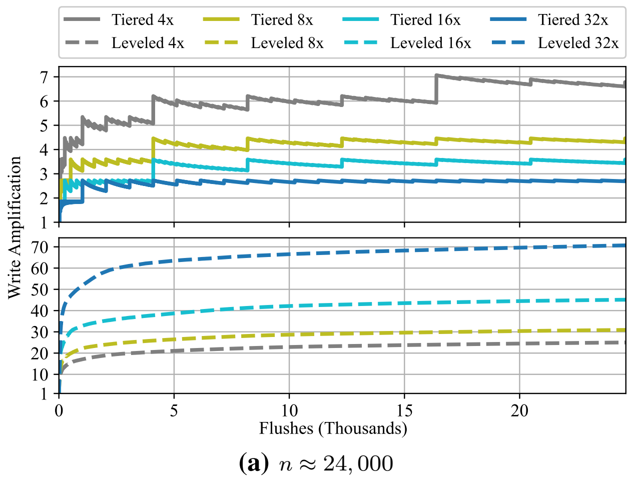
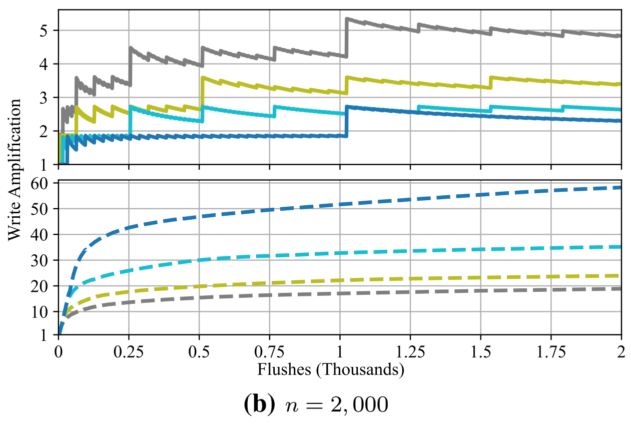



Fig. 4 With $B \in \{4, 8, 16, 32\}$, Tiered are shown as solid lines in the top sub-figures, Leveled are shown as dashed lines in the bottom sub-figures

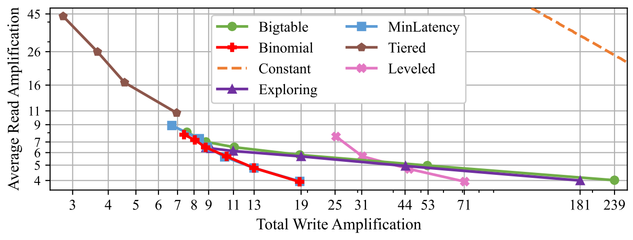


Fig. 5 Average read amplification versus total write amplification (x and y log-scaled)

assuming all metadata and all internal nodes of B^+-trees are cached. Therefore, the *worst-case* read amplification can be computed as the SSTable count for all stack-based policies, or approximately the number of levels for LEVELED policy (number of SSTables in level 0 is 2 in our experiments), although techniques such as Bloom filter can skip checking most of the SSTables, making the actual read amplification be only 1.

As noted previously, MINLATENCY and BIGTABLE, being lazy, tend to keep the read amplification near its limit k. In the large-n regime, any policy that minimizes worst-case write amplification must do this. But, in the small-n regime, BINOMIAL opportunistically achieves smaller average read amplification, as does EXPLORING to some extent.

Figure 5 shows a line for each policy except CONSTANT. The line for CONSTANT was generated from simulation. It can be clearly seen from the figure that CONSTANT is far from optimal; thus, below we concentrate on the other policies. The curve shows the trade-off between final write amplification and average read amplification achieved by its policy: It has a point (x, y) for every run of the bounded depth stack-based policy with $k \in \{4, 5, 6, 7, 8, 10\}$, or TIERED and LEVELED with $B \in \{4, 8, 16, 32\}$ (and $n \approx 24{,}000$), where x is the final write amplification for the run and y is the average read amplification over the course of the run. Both the x-axis and y-axis are log-scaled.

First consider the runs with $k \in \{7, 6, 5, 4\}$. Within each curve, these correspond to the *four rightmost/lowest points* (with $k = 4$ being rightmost/lowest). These runs are dominated by the large-n regime, and each policy has average SSTable count (y coordinate) close to k. In this regime, the BINOMIAL and MINLATENCY policies achieve the same (near-optimal) trade-off, while the EXPLORING and BIGTABLE policies are far from the optimal frontier due to their larger write amplification.

Next consider the remaining runs, for $k \in \{10, 8\}$. On each curve, these are the two leftmost/highest points, with $k = 10$ leftmost. In the curve for EXPLORING, its two points are indistinguishable. These runs stay within the small-n regime. In this regime, BINOMIAL achieves a slightly better trade-off than the other policies. MINLATENCY and BIGTABLE give comparable trade-offs. For EXPLORING, its two runs lie close to the optimal frontier, but low: increasing the SSTable limit (k) from 8 to 10 makes little difference.

The read amplification of TIERED and LEVELED is inverse of their write amplification, that is $O(B \log t)$ for TIERED and $O(\log t)$ for LEVELED. TIERED—the 4 points from left to right correspond to $B \in \{32, 16, 8, 4\}$, respectively—has higher read amplification than any other policy tested but has significantly lower write amplification. LEVELED—the 4 points from left to right correspond to $B \in \{4, 8, 16, 32\}$, respectively—has comparable read amplification to the other policies except TIERED, but has much higher write amplification. Usually, a merge policy cannot achieve low write and read amplification at the same time. Most researches tried to improve the trade-off curve of TIERED and LEVELED such that it can get closer to the optimal frontier [13,14]. As shown in the figure, BINOMIAL and MINLATENCY are both closer to the optimal frontier, which has better trade-off between write and read. BIGTABLE and EXPLORING are closer to the optimal frontier with a few large k values, but they have to pay very high write cost to reduce the read cost.

4.3.3 Transient space amplification

Recently, various works [7,16,28] have discussed the importance of *space amplification*. In an LSM-tree based database system, space amplification is mostly determined by the amount of obsolete data from updates and deletions in a stable state which are yet to be garbage-collected in merges. Because our primary focus is an append-only workload without any updates or deletions, there would be no obsolete data, so the space amplification of all policies would be almost the same. Therefore, comparing space amplification among these policies is not interesting here.

On the other hand, what is more interesting is the *transient space amplification*, which measures the temporary disk space required for creating new components [18,33] during merges. We compute transient space amplification as the maximum total size of all SSTables divided by the total data size flushed (inserted) so far. For example, a flush in TIERED or LEVELED can trigger several merges in sequence, where only the largest merge will be counted. A maximum of transient space amplification of 2 can happen when a major merge involves all existing SSTables. A policy with higher transient space amplification needs larger disk space to load the same amount of data, causing lower disk space utilization. The highest transient space amplification observed in our experiments for each policy is shown in Fig. 6, where BINOMIAL, MINLATENCY, BIGTABLE, and EXPLORING use $k = 4$, TIERED uses $B = 4$, and LEVELED uses $B = 32$. All stack-based policies tested (including TIERED) could eventually reach a transient space amplification of 2, while LEVELED has very low transient space amplification that is close to 1, and hence utilizes disk space much better. Among the five

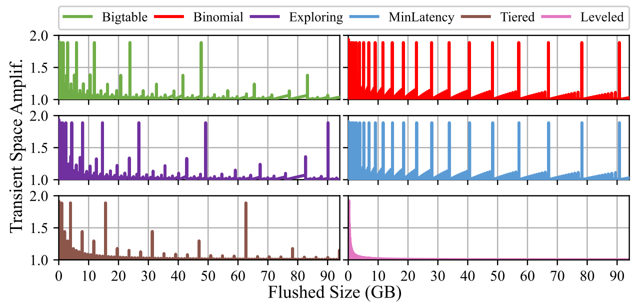
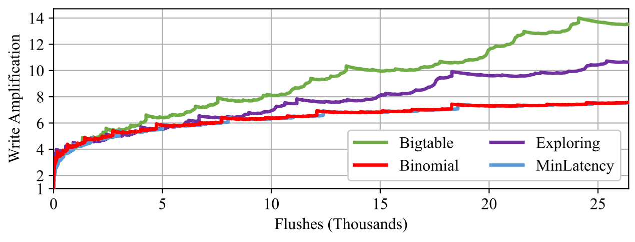
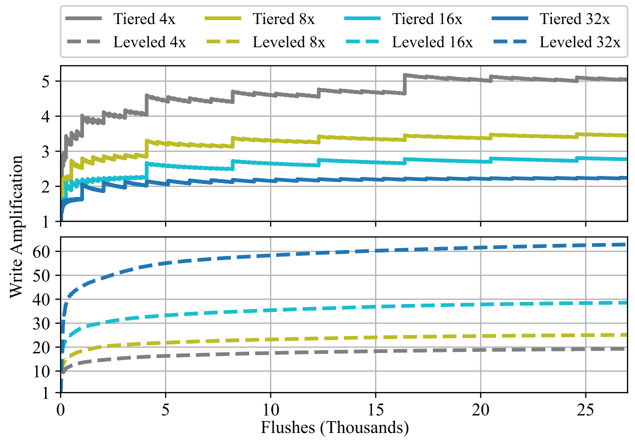


Fig. 6 Transient space amplification of stack-based policies with $k = 4$ or $B = 4$ and LEVELED with $B = 32$



(a) Four bounded-depth policies with $k = 6$



(b) TIERED and LEVELED with $B \in \{4, 8, 16, 32\}$.

Fig. 7 Runs with random UPDATEs ($n \approx 26,400$)

stack-based policies, TIERED offered lowest transient space amplification but highest read amplification; the transient space amplification is lower than the others for BIGTABLE, but its write amplification is high. But in general, any policy which tries to reduce the total number of SSTables to a minimum can have a high transient space amplification close to 2 due to major merges.

4.4 Updates and deletions

UPDATE and DELETE operations insert records whose keys already occur in some SSTable. As a merge combines SSTables, if it finds multiple records with the same key, it can remove all but the latest one. Hence, for workloads with UPDATE and DELETE operations, the write amplification can be reduced. But the experimental runs described above have no UPDATE or DELETE operations. As a step toward understanding their effects, we did additional runs with $k = 6$ and $B \in \{4, 8, 16, 32\}$, with 70% of the WRITE operations replaced by UPDATEs, each to a key selected randomly from the existing keys according to a Zipf distribution with exponent $E = 0.99$, concentrated on the recently inserted items, similar to the "Latest" distribution in YCSB. The flush rate is reduced, as UPDATEs to keys currently in the MemTable are common but do not increase the number of records in the MemTable. To compensate, we increased the total number of operations by 50%, resulting in about $n \approx 26,400$ flushes.

Figure 7a, b plots the write amplification versus flushes for the 4 runs of the bounded depth policies and for the 8 runs of TIERED and LEVELED, respectively.

The primary effect (not seen in the plots) is a reduction in the total number of flushes, but the write amplification (even as a function of the number of flushes) is also somewhat reduced, compared to the original runs (Fig. 3b). The relative performance of the various policies is unchanged. Experiments with other key distributions (uniform over existing keys, or Zipf concentrated on the oldest) yielded similar results that we do not report here.

Although the theoretical model mostly focuses on the append-only workload, via the experiments shown in Fig. 7, the write amplifications are still aligned with the model's prediction even with a update-heavy workload. In practice, with more updates or deletions, the error between the theoretical and the actual write cost of BINOMIAL and MINLATENCY can become more and more significant. One way to solve this problem is to periodically re-evaluate the current status of SSTables and recompute the number of flushes based on the total SSTable size. Hence, BINOMIAL and MINLATENCY are still good candidates even for update-heavy workloads in real-world applications.

4.5 Insertion order

Unlike stack-based policies where merges are independent of SSTable contents, LEVELED is highly sensitive to the insertion order, which affects the number of overlapping SSTables in every merge. For workloads with sequentially inserted keys, SSTables do not overlap with each other, and hence they are simply moved to the next level by updating only the metadata with no data copy [38]. For an append-only workload with sequential insertion order, LEVELED can achieve a minimum write amplification that is close to 1, as all merges are just movements of SSTables. However, if updates or deletions were added to such workload, we found that LEVELED could have very similar write amplification as a workload with non-sequential insertion order. We reran the same exper-

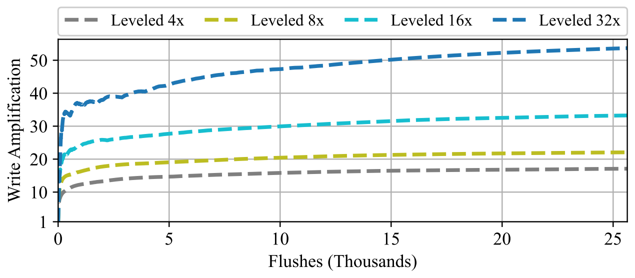


Fig. 8 Runs of sequential insertion with random UPDATEs for LEVELED ($n \approx 26,400$)

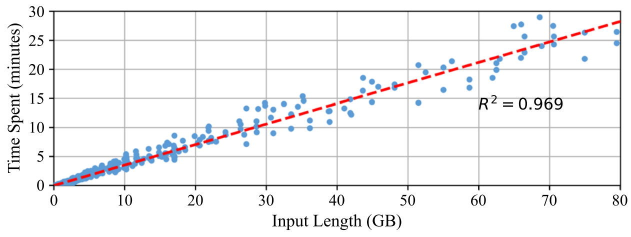
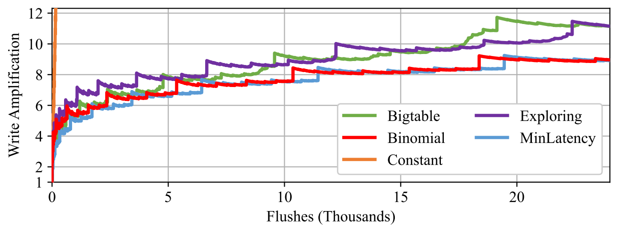
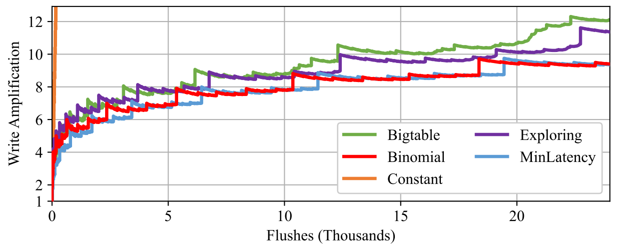


Fig. 9 Time for each merge versus bytes written



(a) Observed write amplification over time. ($k = 7, n = 24,000$)



(b) Simulated write amplification over time. ($k = 7, n = 24,000$)

Fig. 10 Observed and simulated write amplification over time

iments for LEVELED with updates, except that we changed the insert order from *hashed* to *ordered*, such that new keys inserted are in sequential order, while some of the inserted keys can be updated later. Results of these runs are shown in Fig. 8, which is almost identical to Fig. 7b. The insertion order does not impact the write amplification of the other stack-based policies by much; thus, we do not report their write amplifications here.

A minor observation from these runs is, compared to the runs with *hashed* insertion order, the total number of flushes is slightly reduced, leading to a slightly lower write amplification. This is because AsterixDB implements MemTable as B^+-tree. The MemTables are usually 1/2 to 2/3 full with *hashed*, so flushes are triggered more often. As SSTables are created using bulk loading method, their B^+-tree fill factors are very high, making the flushed SSTable size smaller than the MemTable size. For a workload with sequential insertion order, both MemTables and SSTables have very high fill factors, so flushes are triggered less frequently.

5 Model validation and simulation

In each run, the *total time* spent in each merge operation is well predicted by the bytes written. This is demonstrated by the plot in Fig. 9, which has a point for *every individual merge* in every run, showing the elapsed time for that merge versus the number of bytes written by that merge.

Also, observed write amplification is in turn well predicted by the theoretical model. More specifically, using the assumptions of the theoretical model, we implemented a simulator [42] that can simulate a given policy on a given input for any stack-based policies. For each of the 35 runs of the bounded depth policies from the experiment, we simulated the run (for the given policy, k, and $n \approx 24{,}000$ uniform flushes).

Figure 10a illustrates the five runs with $k = 7$, over time, by showing the write amplifications over time as observed in the *actual* runs. Figure 10b shows the same for the *simulated* runs. For the static policies MINLATENCY and BINOMIAL, the observed write amplification tracks the predicted write amplification closely. For EXPLORING and BIGTABLE, the observed write amplification tracks the predicted write amplification, but not as closely. (For these policies, small perturbations in the flush sizes can affect total write amplification.)

Figure 11 shows that the simulated write amplification is a reasonable predictor of the write amplification observed empirically. That figure has two plots. The first (top) plot in that figure has a curve for each policy (except CONSTANT), with a point for each of the six or seven runs that the policy completed, showing the observed final write amplification versus the simulated final write amplification. The two extreme points in the upper right are for BIGTABLE and EXPLORING with $k = 4$, with very high write amplification. To better show the remaining data, the second (bottom) plot expands the first, zooming in to the lower left corner (the region with $x \in [7, 39]$). For each curve, the R^2 value of the best-fit linear trendline is shown in the upper left of the first plot. (The trendlines are not shown.) The R^2 values are very close to 1, demonstrating that the simulated write amplifica-

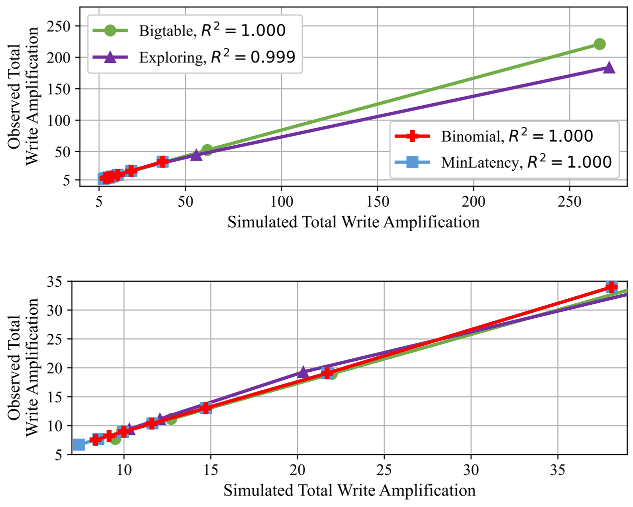


Fig. 11 Observed versus simulated write amplification. The bottom plot zooms in to the portion with $x \in [7, 39]$

tion is a good predictor of the experimentally observed write amplification.

Policy design via analysis and simulation

A realistic theoretical model facilitates design in at least two ways. As described earlier, the model allows a precise theoretical analysis of the underlying combinatorics, as illustrated by the design of MINLATENCY and BINOMIAL. It also allows accurate simulation. As noted in the Introduction, LSM systems are designed to run for months, incorporating terabytes of data. Even with appropriate adaptations, real-world experiments can take days or weeks. Replacing experiments by (much faster) simulations can moderate this bottleneck. As a proof of concept, Fig. 12 shows *simulated* write amplification over time for BIGTABLE, BINOMIAL, CONSTANT, EXPLORING, and MINLATENCY for $k \in \{6, 7, 10\}$. As these policies' average read amplification are all less than k ($\frac{k}{2}$ for CONSTANT), the following settings were used for TIERED and LEVELED to achieve similar average read amplification (assuming one SSTable overlaps with B SSTables in the next level in every merge for LEVELED):

- Figure 12a: $k = 6$, $n = 100{,}000$, $B = 2$ for TIERED and $B = 9$ for LEVELED, their average read amplification are 8.15 and 6.25, respectively;
- Figure 12b: $k = 7$, $n = 100{,}000$, $B = 2$ for TIERED and $B = 6$ for LEVELED, their average read amplification are 8.15 and 7.33, respectively;
- Figure 12c: $k = 10$, $n = 1{,}000{,}000$, $B = 2$ for TIERED and $B = 9$ for LEVELED, their average read amplification are 9.88 and 10.53, respectively.

The smallest size ratio allowed for TIERED is 2, which also provides the lowest average read amplification it can achieve. In all settings, the write amplification of LEVELED is 3 to 4 times larger than BINOMIAL and MINLATENCY; they are only comparable with EXPLORING and slightly better than BIGTABLE in the first plot. TIERED, on the other hand, had lower write amplification than BIGTABLE and EXPLORING, but always higher than BINOMIAL and MINLATENCY, and its average read amplification is higher too, except for the last plot, which is slightly lower than 10.

These simulations took only minutes to complete.

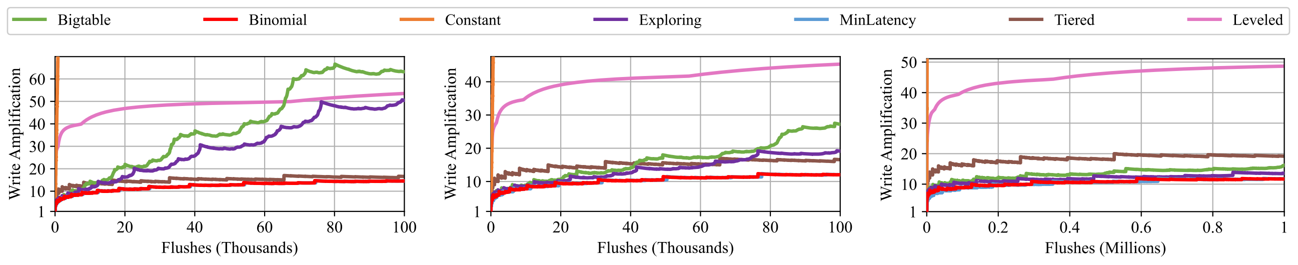
6 Discussion

As predicted by the theoretical model, policy behavior fell into two regimes: the *small-n* regime (until the number of flushes reached about 4^k) and the *large-n* regime (after that). MINLATENCY achieved the lowest write amplification, with BINOMIAL a close second to it. BIGTABLE and EXPLORING were not far behind in the small-n regime, but in the large-n regime their write amplification was an order of magnitude higher. In short, the two newly proposed policies achieve near-optimal worst-case write amplification among all stack-based policies, outperforming policies in use in industrial systems, especially for runs with many flushes.

The trade-offs between write amplification and average read amplification were also studied in this paper. In the large-n regime, all bounded depth policies except (sometimes) EXPLORING had average read amplification near k. MINLATENCY and BINOMIAL, but not EXPLORING or BIGTABLE, were near the optimal frontier. In the small-n regime, all policies were close to the optimal frontier, with BINOMIAL and EXPLORING having average read amplification below k. On the other hand, although popular in the literature, the trade-offs of TIERED and LEVELED are much worse than BINOMIAL and MINLATENCY. These two policies might be overrated if we focus more on the cost of writes and reads.

Limitations and future work

Non-uniform flush sizes, UPDATE*s, dynamic policies.* The experiments here are limited to near-uniform inputs, where most flush sizes are about the same. Most LSM database systems, including AsterixDB, Cassandra, HBase, LevelDB, and RocksDB, use uniform flush size. Some of them support checkpointing, which flushes the MemTable at some timeout interval, or when the commit log is full, potentially creating smaller SSTable before the MemTable is full. Although uniform or near-uniform flush is more common in the literature, workloads with variable flush sizes are of interest. Variable flush size may be used to coordinate multiple datasets sharing the same memory budget, or balance the write buffer and the read buffer cache for dynamic workloads. For example, a recent work [32] described an architecture which provides adaptive memory management to minimize



(a) $k = 6$ (TIERED: $B = 2$, LEVELED: $B = 9$), $n = 100,000$; $4^k = 4,096$.

(b) $k = 7$ (TIERED: $B = 2$, LEVELED: $B = 6$), $n = 100,000$; $4^k = 16,384$.

(c) $k = 10$ (TIERED: $B = 2$, LEVELED: $B = 4$), $n = 1,000,000$; $4^k \approx 10^6$.

Fig. 12 Simulated total write amplification for 100,000 flushes (and 1,000,000 for $k = 10$)

overall write costs. For moderately variable flush sizes, we expect the write amplification of most policies (and certainly of MINLATENCY and BINOMIAL) to be similar to that for uniform inputs. Regardless of variation, the write amplification incurred by MINLATENCY and BINOMIAL is guaranteed to be no worse than it is for uniform inputs.

Most of the experiments here are limited to APPEND-only workloads. A few preliminary results here suggest that a moderate to high rate of UPDATEs and DELETEs mainly reduce flush rate and slightly reduce write amplification. At a minimum, UPDATEs and DELETEs are guaranteed not to increase the write amplification incurred by MINLATENCY and BINOMIAL. But inputs with UPDATEs, DELETEs, and non-uniform flush sizes can have optimal write amplification substantially below the worst case of $\Theta(n^{1/k})$. In this case, dynamic policies such as BIGTABLE, EXPLORING, and new policies which are designed using the theoretical framework of *competitive-analysis* (as in [35]), may, in principle, outperform static policies such as BINOMIAL and MINLATENCY. Future work will explore how significant this effect may be in practice. On the other hand, all the six evaluated stack-based policies are not very sensitive to workloads with UPDATEs or DELETEs; their relative ranking of write and read cost almost remains the same. The exception is LEVELED, which is very sensitive to UPDATEs or DELETEs if the insertion order is nearly sequential. With a very low rate of UPDATEs or DELETEs, write amplification of LEVELED increased significantly.

Compression Many databases support data compression to reduce the data size on disk, at a cost of higher CPU usage to retrieve data with decompression. In general, a system with stronger compression has lower write amplification and space amplification because of smaller data size [16]. Moreover, compression makes flushed SSTable size smaller than the MemTable size which can potentially affect the performance of BINOMIAL and MINLATENCY.

Read costs The experimental design here focuses on minimizing write amplification, relying on the bounded depth constraint to control read costs such as *read amplification*—the average number of disk accesses required per read for point queries. Most LSM systems (other than Bigtable) offer merge policies that are not depth-bounded, instead allowing the SSTable count to grow, say, logarithmically with the number of flushes. A natural objective would be to minimize a linear combination of the read and write amplification—this could control the stack depth without manual configuration. (This is similar to BINOMIAL's behavior in the small-n regime, where it minimizes the worst-case maximum of the read and write amplification, achieving a reasonable balance.) For read costs, a more nuanced accounting is desirable: It would be useful to take into account the effects of Bloom filters, and dynamic policies that respond to varying *rates* of READs are also of interest. Moreover, read amplification of point queries or range queries, query response time and throughput, sequential versus random access to SSTables can be of interest as well.

Secondary indexes The existence of secondary indexes impacts merging. For example, AsterixDB (with default settings) maintains a one-to-one correspondence between the SSTables for the primary indexes and the SSTables for the secondary indexes. Ideally, merge policies should take secondary indexes into account.

7 Related work

Historically, the main data structure used for on-disk key value storage is the B^+-tree. Nonetheless, LSM architectures are becoming common in industrial settings. This is partly because they offer substantially better performance for write-heavy workloads [24]. Further, for many workloads, reads are highly cacheable, making the *effective* workload write-heavy. In these cases, LSM architectures substantially outperform B^+-trees.

In 2006, Google released Bigtable [9,20], now the primary data store for many Google applications. Its default merge policy is a bounded depth stack-based policy. We study it here. Spanner [12], Google's Bigtable replacement, likely uses a stack-based policy, though details are not public.

Apache HBase [5,19,27] was introduced around 2006, modeled on Bigtable, and used by Facebook 2010–2018. Its default merge policy is EXPLORING, the precursor of which was a variant of BIGTABLE called RATIOBASED. Both policies are configurable as bounded depth policies. Here, we report results only for EXPLORING, as it consistently outperformed RATIOBASED.

Apache Cassandra [4,29] was released by Facebook in 2008. Its first main merge policy, SIZETIERED, is a stack-based policy that orders the SSTables by size, groups similar-sized SSTables, and then merges a group that has sufficiently many SSTables. SIZETIERED is not *stable*—that is, it does not maintain the following property at all times: *the* WRITE *times of all items in any given SSTable precede those of all items in every newer SSTable.* With a stable policy, a READ can scan the recently created SSTables first, stopping with the first SSTable that contains the key. Unstable policies lack this advantage: A READ operation must check *every* SSTable. Apache Accumulo [26] which was created in 2008 by the NSA uses a similar stack-based policy. We do not test these policies here, as our test platform supports only stable policies, and we believe they behave similarly to BIGTABLE or EXPLORING.

Previous to this work, our test platform—Apache AsterixDB—provided just one bounded depth policy (CONSTANT), which suffered from high write amplification [3]. AsterixDB has removed support for CONSTANT, and, based on the preliminary results provided here, added support for BINOMIAL. Our recent work [34] shows that BINOMIAL can provide superior write and read performance for LSM secondary spatial indexes, too.

Leveled policies LevelDB [15,21] was released in 2011 by Google. Its merge policy, unlike the policies mentioned above, does not fit the stack-based model. For our purposes, the policy can be viewed as a modified stack-based policy where each SSTable is split (by partitioning the key space into disjoint intervals) into multiple smaller SSTables that are collectively called a *level* (or *sorted run*). Each READ operation needs to check only one SSTable per level—the one whose key interval contains the given key. Using many smaller tables allows smaller, "rolling" merges, avoiding the occasional monolithic merges required by stack-based policies.

In 2011, Apache Cassandra added support for a leveled policy adapted from LevelDB. (Cassandra also offers merge policies specifically designed for time-series workloads.) In 2012, Facebook released a LevelDB fork called RocksDB [16,17]. RocksDB offers several policies: the standard TIERED and LEVELED, LEVELED-N which allows multiple sorted runs per level, a hybrid of TIERED+LEVELED, and FIFO which aims for cache-like data [18].

Mixed of tiered and leveled policies In the literature, TIERED provides very low write amplification but very high read amplification. On the other hand, LEVELED provides good read amplification at the cost of high write amplification. It is natural to combine these 2 policies together to achieve a more balanced trade-off between write and read amplification. In RocksDB [16,17], there are 2 policies of such mix of TIERED and LEVELED policies. The Leveled-N policy allows N sorted run in a single level instead of 1 sorted run per level. Similar idea was also described in [13,14]. The other policy is called tiered+leveled, which uses TIERED for the smaller levels and LEVELED for the larger levels. This policy allows transition from TIERED to LEVELED at a certain level. SlimDB [39] is one example of this policy. It is an interesting research direction to evaluate and compare their trade-offs between write and average read amplification with BINOMIAL and MINLATENCY.

None of the leveled or mixed policies are stack-based or bounded depth policies.

Other merge policy models and optimizations Independently of Mathieu et al. [35], Lim et al. [30] propose a similar theoretical model for write amplification and point out its utility for simulation. The model includes a statistical estimate of the effects of for UPDATEs and DELETEs. For leveled policies, Lim et al. use their model to propose tuning various policy parameters—such as the size of each level—to optimize performance. Dayan et al. [13,14] propose further optimizations of SIZETIERED and leveled policies by tuning aspects such as the Bloom filters' false positive rate (vs. size) according to SSTable size, the per-level merge frequency, and the memory allocation between buffers and Bloom filters.

Multi-threaded merges (exploiting SSD parallelism) are studied in [10,16,31,43]. Cache optimization in leveled merges is studied in [41]. Offloading merges to another server is studied in [1].

Some of the methods above optimize READ performance; those complement the optimization of write amplification considered here. None of the above works consider bounded depth policies.

This paper focuses primarily on write amplification (and to some extent read amplification). Other aspects of LSM performance, such as I/O throughput, can also be affected by merge policies but are not discussed here. For a more detailed discussion of LSM architectures, including compaction policies, see [33].

8 Conclusions

This work compares several bounded depth LSM merge policies, including representative policies from industrial NoSQL databases and two new ones based on recent theoretical modeling, as well as the standard TIERED policy and LEVELED policy, on a common platform (AsterixDB) using Yahoo! cloud serving benchmark. The results have validated the proposed theoretical model and show that, compared to existing policies, the newly proposed policies can have substantially lower write amplification. TIERED and LEVELED, while popular in the literature, generally underperform because of their worse trade-off between writes and reads. The theoretical model is realistic and can be used, via both analysis and simulation, for the effective design and analysis of merge policies. For example, we shared our experimental findings with the developers of Apache AsterixDB [6], and BINOMIAL, designed via the theoretical model, has now been added as an LSM merging policy option to AsterixDB.

Acknowledgements We would like to thank the AsterixDB team for their help with the AsterixDB internals, and C. Staelin and A. Yousefi from Google for discussions about Bigtable. This work was supported by NSF Grants IIS-1619463, IIS-1838222, IIS-1901379 and Google research award titled: "A Study of Online Bigtable-Compaction Algorithms."

A Appendix (Data)

For each of the 43 runs, Table 1 shows the total write amplification and the average read amplification at five points during the run: after 1000, 3000, 5000, 10,000, and 20,000 flushes. If it happens that the MemTable is flushed while a merge is ongoing, the SSTable count may briefly exceed k. For this reason, the average read amplification slightly exceeded k in a few runs (with $k \in \{3, 4, 5\}$—see the bold numbers).

Table 1 Total observed write amplification and average read amplification for all runs, for various n

Policy		$n = 1000$		3000		5000		10,000		20,000	
	k/B	Write amplif.	Read amplif.	Write amplif.	Read amplif.	Write amplif.	Read amplif.	Write amplif.	Read amplif.	Write amplif.	Read amplif.
Bigtable	3	37.57	**3.48**	87.37	**3.71**	165.89	**3.79**	N/A			
	4	11.64	3.97	31.89	**4.35**	46.79	**4.46**	86.33	**4.61**	175.06	**4.74**
	5	7.13	4.53	11.19	4.79	15.80	4.96	26.16	**5.17**	46.17	**5.36**
	6	5.78	5.03	7.78	5.36	9.08	5.55	12.52	5.76	18.71	5.96
	7	5.34	5.60	6.69	5.92	7.85	6.10	9.22	6.31	11.46	6.52
	8	5.05	6.00	6.52	6.34	6.52	6.57	7.32	6.80	8.31	7.04
	10	5.31	6.97	5.79	7.39	6.63	7.51	7.26	7.73	7.87	7.98
Binomial	3	12.05	2.95	17.26	**3.01**	20.61	**3.03**	25.72	**3.08**	32.07	**3.13**
	4	8.61	3.71	10.67	3.82	12.72	3.85	14.76	3.91	17.56	3.95
	5	6.38	4.49	8.84	4.65	9.34	4.70	10.86	4.77	12.49	4.83
	6	5.61	5.21	7.33	5.48	8.16	5.57	8.84	5.64	10.34	5.69
	7	5.40	5.39	6.44	6.05	6.77	6.24	7.57	6.40	8.96	6.49
	8	5.40	5.41	6.30	6.21	6.44	6.64	7.37	7.00	7.64	7.23
	10	5.40	5.38	6.30	6.19	6.44	6.62	7.30	7.08	7.19	7.68
Exploring	3	30.67	**3.31**	94.57	**3.63**	164.31	**3.76**	N/A			
	4	12.52	3.80	22.88	**4.07**	34.83	**4.23**	68.71	**4.43**	153.02	**4.61**
	5	7.00	4.31	10.55	4.66	13.08	4.79	21.72	**5.06**	37.00	**5.26**
	6	6.20	4.51	7.68	5.06	8.75	5.23	11.22	5.50	16.93	5.77
	7	5.99	4.58	7.41	5.19	7.73	5.42	8.66	5.81	10.07	6.12
	8	5.97	4.56	7.12	5.20	7.45	5.47	8.19	5.91	8.71	6.33
	10	5.90	4.55	6.99	5.19	7.33	5.48	7.98	5.97	8.37	6.40
MinLatency	3	12.10	**3.00**	17.26	**3.03**	20.62	**3.04**	25.72	**3.08**	32.07	**3.12**
	4	7.89	3.73	10.68	3.84	12.74	3.88	14.79	3.94	17.58	3.98
	5	6.38	4.51	8.11	4.66	9.37	4.70	10.90	4.76	12.48	4.82
	6	5.86	5.24	6.75	5.46	7.52	5.52	8.78	5.58	10.41	5.64
	7	4.97	6.09	5.96	6.30	6.59	6.37	7.55	6.44	9.13	6.51
	8	4.34	6.77	5.30	7.10	6.03	7.13	6.89	7.24	7.64	7.33
	10	3.69	8.24	4.52	8.56	5.03	8.63	5.87	8.78	6.93	8.91
Tiered	4	4.25	8.36	5.01	9.39	5.87	9.84	5.98	10.53	6.73	11.34
	8	3.15	11.75	3.46	13.81	4.24	14.54	4.28	15.31	4.30	16.86
	16	2.52	17.94	2.67	22.05	3.41	23.35	3.43	24.37	3.43	26.19
	32	1.86	31.54	2.45	33.24	2.57	34.36	2.66	36.95	2.70	41.98
Leveled	4	17.05	5.36	19.87	6.13	21.07	6.48	22.87	6.95	24.48	7.48
	8	22.17	4.05	25.03	4.68	26.47	4.81	28.66	5.23	30.37	5.62
	16	32.75	3.59	36.60	3.86	38.68	3.92	42.15	4.33	44.47	4.67
	32	51.65	2.95	60.98	3.48	63.50	3.69	66.54	3.84	69.62	3.92

References

1. Ahmad, M.Y., Kemme, B.: Compaction management in distributed key-value datastores. Proc. VLDB Endow. **8**(8), 850–861 (2015)
2. Alsubaiee, S., Altowim, Y., Altwaijry, H., Behm, A., Borkar, V., Bu, Y., Carey, M., Cetindil, I., Cheelangi, M., Faraaz, K., Gabrielova, E., Grover, R., Heilbron, Z., Kim, Y.S., Li, C., Li, G., Ok, J.M., Onose, N., Pirzadeh, P., Tsotras, V., Vernica, R., Wen, J., Westmann, T.: AsterixDB: a scalable, open source BDMS. Proc. VLDB Endow. **7**(14), 1905–1916 (2014a)
3. Alsubaiee, S., Behm, A., Borkar, V., Heilbron, Z., Kim, Y.S., Carey, M.J., Dreseler, M., Li, C.: Storage management in AsterixDB. Proc. VLDB Endow. **7**(10), 841–852 (2014b)
4. Apache Software Foundation: Apache Cassandra. (2019a). http://cassandra.apache.org
5. Apache Software Foundation: Apache HBase. (2019b). https://hbase.apache.org
6. Apache Software Foundation: Apache AsterixDB. (2020). https://asterixdb.apache.org
7. Balmau, O., Dinu, F., Zwaenepoel, W., Gupta, K., Chandhiramoorthi, R., Didona, D.: SILK: Preventing latency spikes in log-structured merge key-value stores. In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), pp 753–766 (2019)
8. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec. **39**(4), 12–27 (2011)
9. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
10. Chen, F., Lee, R., Zhang, X.: Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In: 2011 IEEE 17th International Symposium on High Performance Computer Architecture, pp. 266–277. IEEE (2011)
11. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, ACM, SoCC '10, pp. 143–154 (2010)
12. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8:1–8:22 (2013)
13. Dayan, N., Idreos, S.: Dostoevsky: Better space-time trade-offs for LSM-Tree based key-value stores via adaptive removal of superfluous merging. In: Proceedings of the 2018 International Conference on Management of Data, ACM, SIGMOD '18, pp. 505–520 (2018)
14. Dayan, N., Athanassoulis, M., Idreos, S.: Monkey: Optimal navigable key-value store. In: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD, pp. 79–94 (2017)
15. Dent, A.: Getting started with LevelDB. Packt Publishing Ltd, London (2013)
16. Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M.: Optimizing space amplification in RocksDB. In: CIDR, CIDR, vol 3, p. 3 (2017)
17. Facebook, Inc: RocksDB. (2020a). https://rocksdb.org
18. Facebook, Inc: RocksDB Wiki: Compaction. https://github.com/facebook/rocksdb/wiki/Compaction (2020b)
19. George, L.: HBase: The Definitive Guide: Random Access to Your Planet-Size Data. O'Reilly Media Inc, Newton (2011)
20. Google LLC: Bigtable. (2019a). https://cloud.google.com/bigtable
21. Google LLC: LevelDB. (2019b). https://github.com/google/leveldb
22. Graefe, G., et al.: Modern B-tree techniques. Found. Trends Databases **3**(4), 203–402 (2011)
23. Grover, R., Carey, M.J.: Data ingestion in AsterixDB. In: EDBT, OpenProceedings.org, pp. 605–616 (2015)
24. Jermaine, C., Omiecinski, E., Yee, W.G.: The partitioned exponential file for database storage management. VLDB J. **16**(4), 417–437 (2007)
25. Judd, D.: Scale out with HyperTable. Linux magazine, August 7th 1, (2008). http://www.linux-mag.com/id/6645
26. Kepner, J., Arcand, W., Bestor, D., Bergeron, B., Byun, C., Gadepally, V., Hubbell, M., Michaleas, P., Mullen, J., Prout, A., et al.: Achieving 100,000,000 database inserts per second using Accumulo and D4M. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE, IEEE (2014)
27. Khetrapal, A., Ganesh, V.: HBase and Hypertable for large scale distributed storage systems. Department of Computer Science, Purdue University 10(1376616.1376726) (2006)
28. Kuszmaul, B.C.: A comparison of fractal trees to log-structured merge (LSM) trees. Tokutek White Paper (2014)
29. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
30. Lim, H., Andersen, D.G., Kaminsky, M.: Towards accurate and fast evaluation of multi-stage log-structured designs. In: 14th USENIX Conference on File and Storage Technologies (FAST 16), USENIX Association, pp. 149–166 (2016)
31. Lu, L., Pillai, T.S., Gopalakrishnan, H., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: WiscKey: separating keys from values in SSD-conscious storage. ACM Trans. Storage **13**(1), 5:1–5:28 (2017)
32. Luo, C., Carey, M.J.: Breaking down memory walls: adaptive memory management in LSM-based storage systems (extended version). arXiv:2004.10360 (2020a)
33. Luo, C., Carey, M.J.: LSM-based storage techniques: a survey. VLDB J. **29**(1), 393–418 (2020b)
34. Mao, Q., Qader, M.A., Hristidis, V.: Comprehensive comparison of LSM architectures for spatial data. In: 2020 IEEE International Conference on Big Data (Big Data), pp. 455–460. IEEE (2020)
35. Mathieu, C., Staelin, C., Young, N.E., Yousefi, A.: Bigtable merge compaction. arXiv:1407.3008 (2014)
36. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Inf. **33**(4), 351–385 (1996)
37. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: YCSB++: Benchmarking and performance debugging advanced features in scalable table stores. In: Proceedings of the 2Nd ACM Symposium on Cloud Computing, ACM, SOCC '11, pp. 9:1–9:14 (2011)
38. Raju, P., Kadekodi, R., Chidambaram, V., Abraham, I.: PebblesDB: Building key-value stores using fragmented log-structured merge trees. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp 497–514 (2017)
39. Ren, K., Zheng, Q., Arulraj, J., Gibson, G.: SlimDB: a space-efficient key-value storage engine for semi-sorted data. Proc. VLDB Endow. **10**(13), 2037–2048 (2017)
40. Sears, R., Ramakrishnan, R.: bLSM: a general purpose log structured merge tree. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, SIGMOD '12. pp. 217–228 (2012)
41. Teng, D., Guo, L., Lee, R., Chen, F., Ma, S., Zhang, Y., Zhang, X.: LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In: Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, pp. 68–79. IEEE (2017)
42. UCR DBLab: Merge Simulator. (2019). https://github.com/UC-Riverside-DatabaseLab/MergeSimulator
43. Wang, P., Sun, G., Jiang, S., Ouyang, J., Lin, S., Zhang, C., Cong, J.: An efficient design and implementation of LSM-tree based key-

value store on open-channel SSD. In: Proceedings of the Ninth European Conference on Computer Systems, ACM, EuroSys ’14, pp. 16:1–16:14 (2014)

44. Yahoo!: Yahoo! cloud serving benchmark. (2019). https://github.com/brianfrankcooper/YCSB