How to Write to SSDs

Bohyun Lee
Technische Universität München
bohyun.lee@tum.de

Tobias Ziegler
TigerBeetle
tobias@tigerbeetle.com

Viktor Leis
Technische Universität München
leis@in.tum.de

ABSTRACT

This paper demonstrates that adopting out-of-place writes is essential for database systems to fully leverage SSD performance and extend SSD lifespan. We propose a set of out-of-place optimizations that collectively reduce write amplification across both the DBMS and SSD layers. We redesign the in-place, B-tree-based LeanStore to write out-of-place and support these optimizations, and evaluate it on diverse OLTP benchmarks, dataset sizes, and SSDs. The final design improves throughput by 1.65–2.24× and reduces flash writes per operation by 6.2–9.8× on YCSB-A. On TPC-C with 15,000 warehouses, throughput improves by 2.45× while flash writes decrease by 7.2×. Finally, we show that the architecture can seamlessly support novel SSD interfaces such as ZNS and FDP.

PVLDB Reference Format:
Bohyun Lee, Tobias Ziegler, and Viktor Leis. How to Write to SSDs. PVLDB, 19(7): 1469 - 1483, 2026.
doi:10.14778/3801059.3801063

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/LeeBohyun/ZLeanStore.

1 INTRODUCTION

Modern DBMSs optimize for SSD characteristics. As SSD prices have continued to fall while DRAM costs stagnate, SSDs have replaced disks as the primary storage medium for high-performance DBMSs [63, 73, 83, 85, 121]. Consequently, modern systems increasingly tailor their designs to SSD properties, such as internal parallelism [36], read-write asymmetry [6, 57, 105], and SSD reads [5, 43].
Yet most DBMSs still do not optimize for SSD writes. However, few systems control database *write behavior* to optimize for SSDs, even though SSDs handle writes very differently from hard disks. The overall SSD performance for out-of-memory workloads can be greatly impacted by how a database system writes to the device. Furthermore, SSDs have a finite lifespan: each write wears out flash cells, unlike hard disks, which do not suffer from endurance limits [66, 117]. However, despite these differences, most systems still issue writes as if targeting hard disks – ultimately writing far more than intended while remaining unaware of the consequences.
Database systems write significantly more than expected. Once database writes accumulate beyond a certain threshold, ignoring SSD write characteristics becomes increasingly costly [35]. We benchmark the B-tree-based engine LeanStore [63] on an enterprise SSD filled to 90% using the YCSB-A workload (zipf θ = 0.8), running it until cumulative DB writes exceed four times the SSD's capacity. Surprisingly, as shown in the third bar of the first plot in Figure 1, the in-place-write LeanStore baseline writes 18.85 KiB per 4 KiB B-tree node page write, 4.7× more than expected. A similar trend appears in MySQL and PostgreSQL, as shown in the first two bars.

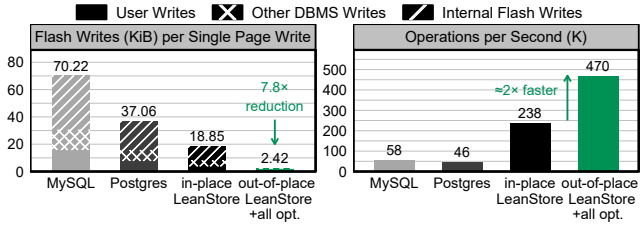


Figure 1: Flash writes per single page write and throughput (YCSB-A zipf θ = 0.8, Samsung PM9A3 SSD 90% full, page sizes (KB): MySQL: 16, PostgreSQL: 8, LeanStore: 4)

Identifying the cause of write amplification. To investigate the source of the amplification, we categorize writes into three types (ignoring WAL writes because they are easy to handle): (1) user writes (including evictions or checkpointing), (2) additional DBMS-issued writes, and (3) SSD-internal writes (obtained by OCP [99]). Here, we identify two main culprits behind this amplification.
Culprit 1: DBMSs themselves write more. The DBMS often issues more writes than expected. When comparing the user writes with the total amount of writes issued by the DBMS (user + other DB-issued writes) in Figure 1, the latter is roughly *double* the user writes due to the doublewrite buffering overhead. This behavior is typical of in-place systems, including MySQL [78] and PostgreSQL (albeit indirectly) [101], as shown in the first two bars in Figure 1.
Culprit 2: SSDs internally amplify DBMS writes. The SSD itself amplifies DBMS writes, a phenomenon known as SSD *write amplification* (WA). In our experiment, LeanStore experiences a Write Amplification Factor (WAF) of 2.36× (Figure 1), meaning total flash writes are 2.36× the DBMS-issued writes. This internal WA degrades performance over time and reduces SSD lifespan [21, 66]. The Samsung PM9A3 data center SSD used in the experiment guarantees a lifetime of 5 years at 1 Drive Write Per Day (DWPD)—corresponding to an average write speed of only 11 MB/s. LeanStore writes about 400 MB/s in the experiment, which means that the SSD would reach its endurance limit in about 1.5 months.
DBMSs taking charge of minimizing end-to-end WA. A single page write is amplified first within the DBMS and then within the SSD, yielding a total WAF of DB WAF × SSD WAF. Conventionally, minimizing SSD WAF is delegated to lower layers—either the SSD or, indirectly, the filesystem [12, 49, 60], while DB WAF is treated separately (e.g., LSM-tree WA optimizations [26, 70]). However, DBMS write issuance dictates both DB and SSD behavior; focusing

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801063

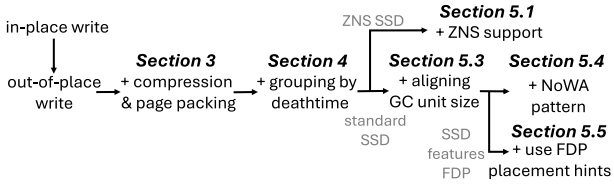
only on DB WAF can paradoxically increase flash writes when SSD WAF is overlooked. Thus, we argue that the DBMS, with the most workload knowledge at the highest layer, should coordinate writes with both DB- and SSD-level WA in mind.

Reducing SSD writes with out-of-place techniques. To minimize flash writes per operation, we first move from in-place to out-of-place updates. By removing the constraint of rewriting a page at a fixed location, the DBMS gains *flexibility* in write placement. We then present several out-of-place-based optimizations that together minimize end-to-end WAF. These are implemented on LeanStore after converting to out-of-place writes. Using the same benchmark, the results (shown in the final bar of the two plots in Figure 1) demonstrate that our out-of-place design with optimizations reduces flash writes by *7.8*× per operation compared to in-place LeanStore. This reduction is achieved by eliminating all other redundant writes, effectively reaching the near-optimal level of user writes alone. Consequently, SSD lifespan is directly extended, while cost, power consumption, and carbon footprint per operation are reduced [7, 90]. Throughput (operations per second, OPS) nearly doubles as well (right panel of Figure 1).

Contributions. The proposed optimizations systematically reduce cross-layer WA by considering how the write pattern shapes DB- and SSD-level amplification and their interplay. Our design requires only modest changes to upper components of the storage engine, including the buffer pool, logging/recovery, and index structure, making it readily applicable to other B-tree-based systems. The proposed optimizations are:

- **Compression & page packing**: Page-wise compression reduces write volume, but with 4 KiB pages, its benefits vanish when compressed pages are misaligned or smaller than 4 KiB. We introduce page packing, grouping compressed pages so each is read with a single 4 KiB access while retaining write and space savings.
- **Grouping by deathtime**: System-level out-of-place writes inherently require garbage collection (GC). To minimize WA during GC, we group pages by their expected invalidation time ("deathtime") when placing them, derived from DB semantics.
- **ZNS support**: Our design naturally extends to Zoned Namespace (ZNS) SSDs, leveraging the existing DB GC and mapping. This enables full compatibility with ZNS, which inherently guarantees an (optimal) SSD WAF of 1.
- **Aligning DB and SSD GC units**: For non-ZNS SSDs, SSD GC-induced WA is mostly inevitable. By estimating the physical placement inside SSDs, we find that the first key to mitigate SSD WAF is to align the DB GC unit with the SSD's internal GC unit. When available, this is achieved using the FDP Reclaim Unit size; otherwise, it can be estimated using a ZNS-like pattern.
- **NoWA pattern**: For commodity non-ZNS SSDs not featuring FDP, we introduce a NoWA (No Write Amplification) pattern that guarantees SSD WAF = 1, even at full device utilization.
- **Using FDP placement hints**: For FDP-enabled SSDs, we show how to use placement hints that can replace the NoWA pattern.

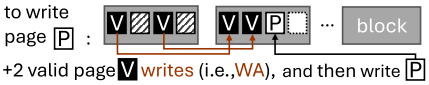
The first two DB WAF optimizations are orthogonal yet complementary, while the remaining SSD-side optimizations may be chosen based on the underlying device. Importantly, applying only the DB WAF optimizations without the SSD WAF ones can increase total WAF; the full set should therefore be viewed jointly. In Section 2, we first motivate why the DBMS is best positioned to optimize SSD writes and why this necessitates out-of-place updates. We then present each optimization as structured in the following figure:



2 THE CASE FOR OUT-OF-PLACE WRITES

We begin by reviewing SSD mechanics and WA, which influence both endurance and throughput. We then argue that the DBMS is best positioned to manage end-to-end WAF at the highest layer. To do so, however, requires adopting out-of-place writes, enabling the DBMS to use workload knowledge and SSD internals to reshape write patterns and thereby minimize flash writes per operation.

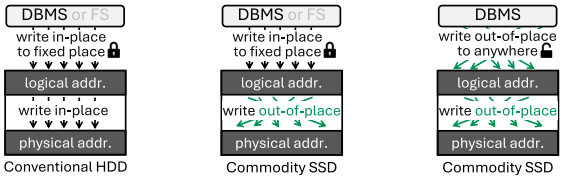
Background: SSD endurance and write amplification. Flash SSDs differ from disks in two properties that matter for write-heavy workloads [57, 110]. First, flash cells only endure a limited number of program/erase cycles; each write gradually wears the cells, so device lifetime is tied to the *volume of physical writes* [15, 66, 113]. Second, SSDs exhibit *write amplification* because writes are performed at the page level (4-16 KiB), whereas erases reclaim entire blocks (many pages). When a block containing still-valid pages must be erased to free space, the SSD first relocates those pages, increasing internal physical writes relative to host writes [45, 47]. Consider a DBMS issuing a write request for page *P*, as illustrated in the figure:



To free space, the SSD's garbage collector selects the first block in the figure as a victim; because two pages in that block remain valid, it copies them elsewhere before erasing the block. Thus, one logical write of page *P* results in three physical writes, directly reducing SSD lifespan and the bandwidth available to the host.

DB WAF: User writes transformed by the DBMS. To understand more about the source of WAs in Figure 1, we first examine the DBMS write path. The DBMS persists a safe page copy before overwriting the target [78, 101], a mechanism known as *doublewrite buffering* (DWB). For this reason, in-place systems such as MySQL, PostgreSQL, XtraDB, and CedarDB incur roughly 2× the logical DB writes due to DWB [74, 86, 103]. Without DWB, the system cannot recover from partial page writes. Some SSDs expose limited atomic-write primitives [50, 119], but typically only for 512- or 4,096-byte units and thus are not widely relied upon.

SSD WAF: DBMS writes transformed by the SSD. Next, depending on the underlying block device, DBMS-issued writes are further amplified inside the device. Here, we assume DBMS is writing directly to the block device without a filesystem, which is common in modern systems [2, 28, 41, 81, 93, 96]. If a filesystem is used, however, it may alter the write pattern the device observes. On disks, logical and physical addresses mostly coincide (Figure 2a); on SSDs, writes are inherently out-of-place (Figure 2b), breaking this fixed



(a) In-place write on a conventional disk **(b) In-place write on a commodity SSD** **(c) Out-of-place write on a commodity SSD**

Figure 2: In-place vs. out-of-place writes on disk and SSD

mapping. As a result, the physical consequences of any given write, particularly its internal placement and resulting WAF, can be severe and difficult to predict, except on specialized devices such as ZNS. For instance, our prior study [35] (Figure 3) shows that several enterprise SSDs unexpectedly exhibit higher WAF (e.g., 4) under even a simple hot/cold workload compared to a uniform-random workload. This demonstrates that SSD WAF is not inherently managed by the device and can vary depending on workload patterns.

Total WAF matters in the end. User writes are first amplified inside the DBMS and then inside the SSD. To quantify this end-to-end amplification, we argue that systems should use *total WAF* as a cross-layer metric. Upon each page write for in-place LeanStore in Figure 1, each logical write to persist a B-tree node is duplicated by DWB (DB WAF = 2.0) and further amplified within the SSD (SSD WAF = 2.36), resulting in the total amplification of 4.7. Thus, workloads experience amplification at both the logical (DB) and physical (SSD) layers, whose combined effect is multiplicative:

$$Total\ WAF = DB\ WAF \times SSD\ WAF$$

DBMS is best positioned to optimize for SSD writes. SSD WAF mitigation has mostly been attempted at lower layers (SSDs or filesystems) [12, 49, 54, 55], but these layers lack workload knowledge and merely receive host writes, leaving them intrinsically unable to change the write pattern that drives SSD WAF. In contrast, *the DBMS controls the write pattern*: by choosing what, when, and where to write (e.g., during eviction or checkpointing), it can proactively influence device behavior. The DBMS possesses richer workload knowledge, including page hotness and access patterns, which are inaccessible to the SSD or filesystem. Thus, with a working understanding of SSD internals, the DBMS can estimate amplification at both levels and adapt its write patterns accordingly.

Out-of-place writes are essential for minimizing total WAF. Even when a system aims to optimize SSD writes, the DBMS can do little if it performs in-place updates. In-place systems fix each page's file location (e.g., using the Page Identifier (PID) as an offset, as in Figure 2b) and overwrite that location in the page size (16 KiB in InnoDB) [31]. Thus, the DBMS cannot alter write placement and therefore *inherits* the device's SSD WAF, forfeiting opportunities to optimize for SSDs. Switching to out-of-place writes (Figure 2c) unlocks flexibility, allowing the DBMS to group and place user writes as it chooses. Out-of-place writes also eliminate doublewrite buffering: the old page remains valid until the new version is durably written, allowing crash recovery via the previous page and WAL replay. Removing DWB nearly halves DB-issued bytes while preserving durability.

Joint consideration of DB and SSD WAF is necessary. To optimize total WAF, the DBMS must account for both DB and SSD WAF and their interplay. Focusing on only one layer can counterintuitively worsen the other. For example, an LSM-tree engine can lower DB WAF by increasing size tiering [23], but doing so consumes more SSD space and can raise SSD WAF, ultimately worsening total WAF. Thus, total WAF—not individual components—must guide optimization, reflecting the effects of DBMS writes at both layers.

Unlocking optimizations that minimize total WAF. In the following sections, we propose several out-of-place-based optimizations that, taken together, minimize total WAF. At the DBMS level, Page-wise Compression (Section 3) and Deathtime-Based GC (Section 4) reduce *DB WAF* by lowering logical write volume. At the device level, we present techniques that ensure *SSD WAF = 1* across all three SSD types: ZNS (Section 5.1), standard (Section 5.4), and standard FDP-enabled SSDs (Section 5.5). To our knowledge, this is the first demonstration of SSD WAF = 1 on commodity SSDs—even at 100% utilization and without fully sequential workloads. Although DB-side and SSD-side techniques must be applied together to minimize total WAF, individual optimizations may be selectively adopted depending on the device (e.g., SSD or disk) and integration layer (e.g., filesystem).

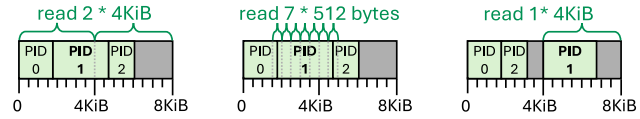
3 REDUCING WRITES WITH COMPRESSION

Compression reduces DB WAF by lowering both the number of bytes written to storage and the overall dataset size. Compressing pages before flushing to the device lowers the logical write volume. However, if the characteristics of the underlying device are ignored, compression can backfire: misaligned or variable-length writes may increase read amplification or fail to reduce write and space at all.

3.1 Page-Wise Compression

How storage-level compression works. Assume a fixed database page size of 4 KiB and a buffer pool that caches at the same granularity. Dirty pages are written to the storage device during eviction or checkpointing. We compress each 4 KiB page *independently* before persistence so it can later be read independently. On reads, the page is decompressed before being placed in the buffer pool.

Compression is difficult to apply with in-place writes. The SSD's device-level write granularity complicates page-level compression for in-place systems. Compressing 4 KiB pages yields variable-length results (typically < 4 KiB), but SSDs and filesystems commonly write in 4 KiB units. If we retain simple $PID \times 4$ KiB addressing and overwrite in place, the device still writes a full 4 KiB block, eliminating any physical savings. One could instead store compressed pages at variable offsets and maintain a PID→offset mapping, but compressed sizes change as values are inserted or updated. Offsets then drift, and relocating a page can cascade into shifting neighboring pages, making updates expensive. Moreover, eviction follows recency rather than physical locality, so pages flushed together are rarely adjacent, hindering coalescing into aligned, sequential writes. Consequently, many in-place DBMSs (e.g., PostgreSQL) forgo page-level compression [87]. Others offload compression to the filesystem (e.g., MySQL) [79]. However, this does not solve the fundamental issue with 4 KiB page sizes, as filesystems also issue I/Os in 4 KiB or larger units.



(a) Read with 4 KiB granularity (b) Read with 512 B granularity (c) Read PID stored by page packing

Figure 3: Total amount of reads when reading PID 1

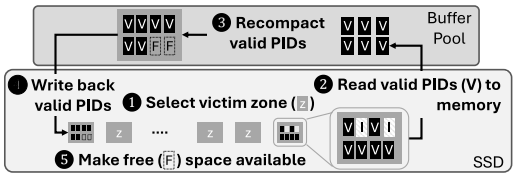


Figure 4: Garbage collection process

Optimization: *easier* compression with out-of-place writes. With out-of-place writes,managing variable-sized compressed pages becomes far more straightforward. For each batch of pages, we compress individual pages, write the resulting compressed batch as a large sequential I/O, and record their new offsets and compressed sizes [109]. This avoids the challenges of in-place systems, making compression far easier to implement.

Compression ratio determines write reduction. How much DB WAF drops depends on data compressibility and the algorithm. We load several datasets into LeanStore and compare compressed vs. uncompressed sizes for TPC-C (1,000 warehouses), YCSB-A (100 GB) [16], and real-world datasets. Using LZ4 [71] and ZSTD [27], compression ratios (as % of original; lower is better) range from 14% to 49%:

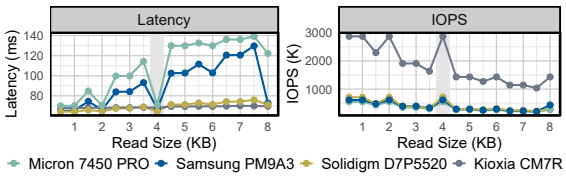
Comp. Ratio (%)	TPC-C	YCSB	Email	URL	Wiki
LZ4 [71]	49.0	41.2	40.1	25.0	31.3
ZSTD [27]	36.5	34.4	27.4	14.5	17.8

These results confirm that most OLTP workloads will benefit from compression, as it reduces write volume and storage footprint.

CPU overheads are negligible in most I/O-bound scenarios. Compression adds CPU work. LZ4, for example, offers a good ratio at high speed [53], needing roughly 2–3 cycles/byte to compress and 0.5–1 cycle/byte to decompress [71]. In most I/O-bound or out-of-memory scenarios, the I/O savings dominate the CPU cost [62]. As we will show in Section 7, compression often improves overall performance, especially in OLTP settings.

3.2 Page Packing After 4 KiB Compression

Read is optimal with 4 KiB granularity and alignment. Read latency strongly influences transaction latency [80]. To identify the optimal read unit, we run an FIO microbenchmark [8] on four enterprise SSDs: random reads from 512 B to 8 KiB in 512 B steps, aligned to request size. Each test ran for 30 minutes, with a single thread for latency (QD=1), and 32 threads (QD=64) for throughput. Across all models, aligned 4 KiB requests achieved the lowest latency and highest throughput:



Smaller or misaligned requests, in contrast, incur additional internal reads [36]. Hence, 4 KiB is the most efficient size and alignment granularity for reading compressed pages.

Read amplification can occur depending on the placement. Using 4 KiB pages with per-page compression yields variable-sized pages that, if poorly placed, can become misaligned on disk. Such misalignment negates compression benefits by preserving write and space costs while introducing read amplification: a single PID read may trigger multiple physical reads [36]. For example, a 3,000 byte compressed page that crosses a 4 KiB boundary requires two 4 KiB reads, totaling ≈ 2.73× its size (Figure 3a). Even with 512 B reads, it still incurs ≈ 1.19× amplification (Figure 3b). This increases read latency and eliminates the intended compression gains.

Optimization: 4 KiB-aligned page packing. To ensure each compressed page is retrievable with a single 4 KiB read, we use *page packing*: align each compressed page to a 4 KiB boundary. When writing a batch (e.g., 64 pages), we compress each page and place it into 4 KiB slots via a best-fit binpacking algorithm [9]. Compressed pages never cross 4 KiB boundaries (Figure 3c). Compared to unaligned 4 KiB or 512 B reads, page packing lets each PID be fetched with one 4 KiB access—cutting I/Os relative to unaligned 4 KiB reads and remaining faster than 512-byte reads. In short, 4 KiB page packing yields fast, predictable reads with minimal amplification (with a small amount of bounded internal slack per page).

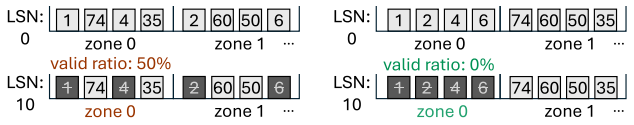
4 DEATHTIME-BASED GC

Out-of-place systems require garbage collection (GC). When a new version of a page is written, the previous version becomes invalid; stale data accumulates until capacity is exhausted. We first explain how GC affects DB WAF, then introduce *Grouping by Death Time* (GDT), which reduces WA for skewed workloads.

4.1 Database-Level Write Amplification and GC

GC overview. GC reclaims space occupied by stale pages, as illustrated in Figure 4. Storage is divided into zones, each containing multiple pages (eight in the figure), defining the GC granularity. When a new page is written, it is appended to a selected zone with available space, and the mapping table is updated while the old version is marked invalid. Over time, zones accumulate a mix of valid and invalid pages. During GC, the garbage collector selects a *victim zone* containing both valid and invalid pages (❶). In Figure 4, 6 of 8 pages are valid, giving a 75% valid ratio. To reclaim space, valid pages not in memory are loaded (❷), compacted (❸), and rewritten (❹). This copyback process reclaims space for two new writes (❺) but incurs WA, as six pages must be rewritten. Garbage-collecting zones with more valid data therefore naturally yields higher WA.

GC valid ratio drives DB WAF. The GC copyback drives DB WAF, as more pages are rewritten than originally requested. WAF depends on the valid ratio of the victim zone in each GC cycle. In Figure 4,



(a) Random placement **(b) Placement using GDT**

Figure 5: Impact of different placement strategies on WA (1 denotes a page with deathtime 1.)

75% of the pages are valid, yielding a WAF of $1/(1-0.75) = 4$, since 75% of the data must be copied to free 25% of the space.

Skewed workloads require better GC. A common GC victim selection (❶ in Figure 4) is greedy; choosing the zone with the fewest valid pages. This is optimal under uniform random workloads [35, 54]. However, real-world workloads are typically skewed [112], making greedy GC suboptimal [35, 49], as skew colocates hot and cold pages within zones, raising valid-page ratios and thus WA.

DBMSs know better. To improve GC under skew, prior work proposes advanced GC and placement schemes—classifying data by hotness [49], expected lifetime [15, 69, 107], or based on spatial and temporal locality [42, 94, 116]. By grouping pages along these dimensions, these methods aim to reduce valid ratios in victim zones. However, these techniques generally target SSD firmware or filesystems, which lack visibility into upper-layer workloads [48, 76, 83], limiting their ability to perform workload-aware GC. In contrast, database systems possess richer workload and data semantics, making them well positioned to exploit intelligent GC schemes [82].

4.2 Grouping Pages By Deathtime

We introduce *Grouping by Death Time* (GDT) [40, 54] into DB GC to reduce DB WAF. GDT uses database semantics to estimate each page's *deathtime* (invalidation time). This classification guides placement and GC, grouping pages that die together into the same zone and thereby reducing WA during reclamation (❸–❻ in Figure 4).

Optimization: placing pages with similar deathtime per zone. GDT packs pages with similar deathtimes into the same zone so they are invalidated together. In Figure 5, eight pages with deathtimes 1, 2, 4, 6, 35, 50, 60, and 74 are written at LSN 0. If placed randomly (Figure 5a), zones mix short- and long-lived pages, yielding a 50% valid ratio at LSN 10 when greedy GC triggers. With GDT (Figure 5b), pages are grouped by deathtime, so at LSN 10, victim zone 0 has no valid pages, eliminating WA for that cycle. Thus, aligning placement with deathtime substantially reduces GC-induced WA.

Using write timestamps for deathtime estimation. To apply GDT, the system must estimate each page's deathtime before writing it to SSD. We extend each page header with a write timestamp, updated whenever the page is persisted [105], and maintain a write history (WH) of the last n timestamps (e.g., 4). GC writes do not update WH. We estimate the next write time as: $EDT = current\_lsn + \frac{WH_n - WH_1}{n-1}$, extrapolating recent write intervals to predict when the page will next be overwritten. For initial writes, pages are grouped by B-tree index ID to colocate pages with similar access patterns [82].

GDT-based placement strategy. When a batch of pages is written upon eviction, GDT first computes each page's EDT. Pages are then grouped and page-packed so that each group contains pages with similar deathtimes. For each group, the strategy selects the zone whose existing pages have the closest average EDT; if no such zone is active, a new zone is opened. This concentrates pages with similar deathtimes into the same zone.

GDT-based GC. GDT placement is effective only when paired with GDT-aware GC; otherwise, GC writes and normal writes may intermix. GC greedily selects victim zones until their cumulative invalid pages free the space of one zone. All valid pages in these victims are loaded, sorted in descending EDT, grouped and packed, and written to an open zone with a matching EDT when possible; otherwise, they remain in their original zone. Pages that are never rewritten (e.g., read-only pages) are assigned the maximum EDT to be treated as the coldest data. This replacement during GC also compensates for initial EDT mispredictions.

Reassigning EDT ranges after GC. After each GC cycle, zones may be full, partially full, or empty. Empty and partially full zones are made available for normal writes. The newly emptied zone is assigned the minimum EDT range, drawing short-lived pages; partially full zones naturally attract warm pages; and full zones retain the coldest pages. This preserves aligned invalidation times within each zone, even after going through GC cycles.

5 MINIMIZING SSD-LEVEL WAF

SSDs also perform out-of-place writes. As discussed in Section 2, SSDs inherently use out-of-place writes and thus incur internal WA. Along with the host write patterns, SSD WAF is shaped by the device's internal architecture and algorithms [65, 66]. Although SSD WAF is critical for performance and lifespan, it remains difficult to mitigate because SSDs operate as black boxes, exposing neither their internal behavior nor physical layout [35]. Moreover, SSDs lack visibility into the workloads running above them.

Different SSD interfaces for minimizing SSD WA. To bridge the knowledge gap between the host and SSD, recent work has explored more communicative SSD interfaces [65]. Zoned Namespace (ZNS) gives the host control over physical layout under certain write constraints, eliminating internal WA [72]. Flexible Data Placement (FDP) allows the host to pass workload hints that the SSD can use to reduce internal WA [117]. While both interfaces are promising, their benefits depend on hardware support and proper integration with host-level management.

Minimizing SSD WA regardless of SSD interfaces. We first show how our out-of-place write design naturally supports ZNS SSDs (Section 5.1). When ZNS is unavailable and SSD-internal GC remains, we align DBMS and SSD GC units (Section 5.3) and introduce the *NoWA write pattern*, which eliminates GC-induced WA even on commodity SSDs (Section 5.4). Finally, for FDP-enabled SSDs, we show how FDP placement hints can substitute for the NoWA pattern (Section 5.5).

5.1 Supporting Zoned Namespace SSDs

SSD WAF = 1 with ZNS. A key benefit of ZNS SSDs is that they eliminate internal WA, making total WAF (DB WAF × SSD WAF) effectively equal to the DB WAF, as GC occurs only at the DB level. To guarantee SSD WAF of 1, ZNS enforces sequential writes

within each zone and shifts GC responsibility to the host. This fits naturally with our design that already uses a translation layer and out-of-place writes with its own GC [12]. With full workload knowledge, GDT-based GC combined with ZNS can perform GC more effectively than in-place systems on commodity SSDs (Figure 2b).

ZNS provides more space and reduces DB WA. ZNS SSDs expose more usable capacity because they require no internal GC and therefore no OP space [12]. Commodity SSDs typically reserve 7–28% of capacity for OP [73], whereas ZNS returns this space to the host [113]. This extra capacity reduces DB WA: from the DB GC's perspective, it acts as additional OP space, allowing GC to be delayed until more pages become invalid. When GC finally runs, zones have lower valid ratios, directly reducing DB WA [21].

Optimization: compatibility with ZNS. To align our layout with ZNS, we set the database zone size (GC unit) to the ZNS zone size. Each zone maintains minimal metadata, including a write pointer and its state (empty, open, full). Writes are issued using zone-append commands to zone IDs. To reuse a full zone after GC copyback, the host issues a zone reset [12]. Overall, our design integrates cleanly with ZNS and remains straightforward to implement.

5.2 Why WA Persists on Commodity SSDs

Standard SSDs can exhibit high WAF. When standard SSDs are used instead of ZNS, the same write pattern can incur high SSD WAF due to internal GC. Although SSD-level WAF reduction has been widely studied, our prior work shows that modern enterprise SSDs still exhibit high WAF even under simple skewed workloads [35]. Thus, even when DB WAF is reduced with our prior optimizations, excessive SSD WAF can negate overall gains.

SSD WAF is shaped by DB GC behavior. For a fixed device, the dominant factor influencing SSD WAF is the write pattern generated by the DBMS [83]. In our setting, DB GC shapes this pattern in three ways. First, GC trigger points determine the SSD's effective OP space: higher fill factors leave less OP space and increase SSD WAF. Second, DB GC granularity affects whether pages from the same zone remain colocated on SSD blocks. Third, victim-zone selection determines which zones are rewritten together, influencing internal write clustering. Together, these choices define the write stream the SSD receives and thus directly affect SSD WAF.

DBMSs have more control than SSDs to mitigate SSD WAF. Although the write pattern is explicit at the DBMS layer, SSDs have no visibility into it [55], making SSD-side mitigation difficult. Prior work attempts to infer workload characteristics inside the SSD [49, 54, 55], but such inference is costly and inherently imperfect. We take the opposite approach: the DBMS, which has full workload knowledge [10], guides writes to reduce SSD WAF directly. By understanding SSD placement and GC behavior, the DBMS can *proactively* structure writes to avoid GC-induced amplification.

5.3 Aligning DB and SSD GC Granularity

A key insight available to the DBMS is that writes to the same zone share similar deathtimes. This section explains why aligning DB and SSD GC granularity is the first step toward reducing SSD WAF using this insight. We then show how the upper bound of SSD GC granularity can be inferred from ZNS-like patterns. When an SSD features FDP, the Reclaim Unit size directly provides this value.

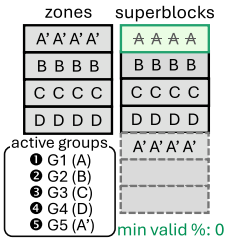


Figure 6: Zone size matches superblock size

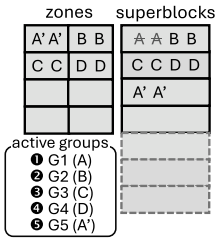


Figure 7: Zone size is smaller than superblock size

Background: placement and GC granularity of SSDs. A superblock is the SSD's primary append unit, formed by grouping erase blocks across planes and dies to exploit internal parallelism. When a stream of page writes arrives, the SSD controller appends them to the active superblock and maps Logical Block Addresses (LBAs) to Physical Block Addresses (PBAs) after persistence. Once the superblock fills, the SSD switches to a new one. As the device fills and free space becomes scarce, GC reclaims space at a vendor-specific granularity, which may range from a single erase block to an entire superblock [66, 117].

Estimating physical placement inside SSDs: simple case. To reason about physical placement, we use a simplified example that reflects general SSD behavior. Suppose the SSD has seven superblocks, each holding four SSD pages (4 KiB), with three reserved as OP space, as shown in Figure 6. The DBMS runs on top of the SSD, and DB GC is invoked when SSD capacity is nearly exhausted (four zones) and stops after reclaiming a target amount of space. An *SSD page* here refers to a 4 KiB LBA offset. SSD pages belonging to the same DB zone share the same label. For example, if zone A spans LBAs 0–16 KiB, its first four SSD pages are labeled *A*. An *active zone* is a zone currently receiving appends, and multiple active zones form an *active group*. A new zone becomes active only after all zones in the current group are fully written. Thus, the number of zones the DBMS appends simultaneously determines the number of active zones and the size of the active group.

Writes from the same zone share the same deathtime. Figure 6 illustrates the case where only one zone is active and the zone size matches the superblock size. When zone A is appended (active group ❶), its four writes (i.e., A pages) are placed together in the first superblock. Later, when DB GC rewrites zone A (active group ❺), the new A' pages are written to the fifth superblock, invalidating all old A pages in the first superblock and leaving it fully invalidated.

DB GC granularity controls deathtime clustering. However, if the zone size is smaller than the superblock size, the first superblock does not become fully invalid. In Figure 7, the zone size is half a superblock. Even if the same active group is used as in Figure 6, SSD pages from zones A and B are interleaved within a superblock, mixing pages with different deathtimes [115]. When DB GC rewrites zone A (❺), only half of the superblock's pages are invalidated.

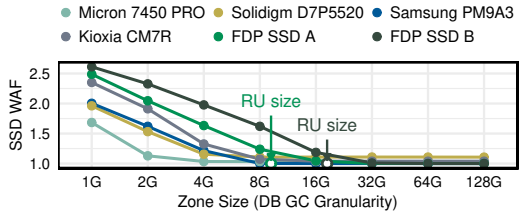
Aligning GC unit size to place writes with the same deathtime. These examples reveal two insights: (1) a write stream whose size matches the superblock is internally placed and GCed together, and (2) writes from a single zone form such a stream only when the DB

and SSD GC units are aligned. Thus, the SSD's GC unit size determines the largest region whose pages can share the same deathtime and yield empty superblocks. From the DBMS's perspective, this maximum size can be used directly as the zone size.

Optimization: using RU size (FDP-enabled SSDs). Standard SSDs usually do not expose the SSD GC granularity. Fortunately, FDP explicitly exposes the SSD's Reclaim Unit (RU), the smallest granularity at which SSD GC operates [117]. With this information, we can set the database's zone size to match the RU size, ensuring that writes are aligned to the SSD's internal GC boundaries.

Optimization: using inferred GC unit size (standard SSDs). For SSDs without FDP, we approximate the upper bound of the SSD's GC unit size using a ZNS-like write pattern. With a single active zone, SSD WAF converges to 1 once the zone size meets or exceeds the internal append unit (Figure 6). Thus, by gradually increasing the zone size and observing when WAF first reaches 1, we estimate this upper bound.

Inferring the GC unit size with a ZNS-like pattern. We validate this approach on six enterprise SSDs, including two FDP-enabled devices, by varying the database zone size:



On FDP-enabled SSD A, WAF drops to 1 at a 16 GB zone size, consistent with its 8,496 MB RU size lying between 8 and 16 GB. The same holds for FDP-enabled SSD B, whose RU size is slightly above 16 GB. Across the other four enterprise SSDs, the inferred GC unit size typically falls between 4 GB and 8 GB. Thus, the zone size at which WAF reaches 1 provides a practical upper bound on SSD GC granularity. Although approximate, this bound is sufficient to avoid mixing DB writes with different deathtimes. For SSDs without OCP commands or FDP support, we recommend using a 32 GB zone size as a safe upper bound. Alternatively, one can indirectly estimate SSD WAF by checking whether throughput drops [35].

5.4 NoWA: SSD WAF=1 on Commodity SSDs

With a single active zone, achieving SSD WAF = 1 is straightforward (Figure 6). With multithreading, however, multiple zones are appended simultaneously, which complicates the problem. In this section, we explain how WA arises from *multiplexing* and *frequency imbalance* among zones within the same active group [83]. We then introduce the NoWA write pattern, which prevents SSD-GC-induced WA and thereby guarantees SSD WAF = 1 on commodity SSDs. For FDP-enabled SSDs, multiplexing can be avoided without NoWA; we discuss this in the next section.

Multiplexing occurs with multiple open zones. When multiple zones are appended concurrently, their write streams interleave across different superblocks [89]. For example, if the DBMS writes to zones A and C at the same time (active group G1(A, C) in Figure 8), writes from both zones become mixed, scattering their data across two superblocks. This multiplexing effect [83] arises because standard SSDs cannot distinguish writes from zones A and C.

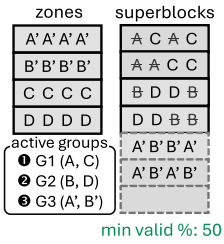


Figure 8: Imbalanced frequency in group ❶ and ❸

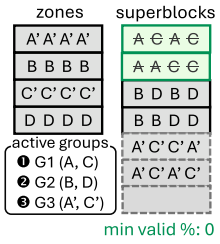


Figure 9: Balanced frequency in all active groups

Frequency imbalance in an active group incurs WA. Multiplexing causes WA when zones in the same active group are invalidated at different rates. In Figure 8, selecting zones A and B for DB GC (active group G3(A', B')) produces four partially invalidated superblocks that the SSD GC must later rewrite. If, however, all zones in the active group are invalidated uniformly, the SSD can still produce superblocks with a 0% valid-page ratio even under multiplexing. For example, choosing G(A', C') in Figure 9 fully invalidates two superblocks. Thus, balanced invalidation within an active group eliminates SSD WA even when writes are multiplexed across zones.

Optimization: NoWA pattern guarantees SSD WAF of 1. The *NoWA pattern* eliminates SSD-GC–induced WA by ensuring that the SSD always has at least one fully invalidated superblock available when it reaches its minimum free–superblock threshold. NoWA achieves this by estimating how pages are multiplexed within the SSD's physical layout and enforcing two rules: (1) defer opening new zones until all currently open zones are completely written, and (2) detect and correct write–frequency imbalances among concurrently appended zones. By selecting DB-GC victim zones in accordance with these principles, NoWA guarantees the existence of an empty superblock before SSD GC is triggered. In effect, this write pattern preempts SSD-level GC.

NoWA properties. The key idea behind NoWA is to eliminate frequency imbalance *before* SSD GC is triggered. As shown in Figure 8, if a new active group (e.g., G3(A', B')) creates uneven invalidation in earlier groups (e.g., A:2 vs. C:1 in G1), the DBMS issues a *compensation write* for the underrepresented zone (e.g., CW(C)), as illustrated in Figure 10. This compensation must occur before the SSD reaches its minimum free–space threshold, which can be estimated using SSD–iq [35]. The compensation write recompacts the scattered pages of zone C into a single superblock, invalidating their older versions and effectively producing two fully invalidated superblocks. Thus, NoWA proactively restores balance whenever imbalance occurs, ensuring that subsequent SSD GC cycles operate on fully invalidated superblocks.

Compensation writes are not entirely redundant. Compensation writes shift a small portion of WA from the SSD to DBMS-level GC. Although they increase DB WAF slightly, the overhead is modest because the DBMS controls zone selection. Before selecting a victim zone, GC can check whether issuing a compensation write would raise valid-page ratios in later rounds; if so, it simply chooses

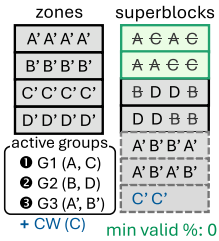


Figure 10: NoWA's compensation writes

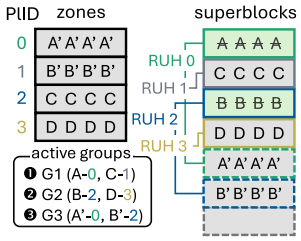


Figure 11: FDP placement hints avoid multiplexing

another zone. Overall, this small increase in DB WAF is far outweighed by the reduction in SSD WAF (e.g., from 2.3 for in-place to 1 with out-of-place LeanStore using NoWA; see Figure 1).

Flexible zone sizing with NoWA. NoWA also allows flexible configuration of DB zone size by adjusting the maximum number of concurrently open zones. NoWA holds as long as max open zones × zone size equals or is a multiple of the inferred SSD GC unit. For example, on FDP-enabled SSD A, 16 active zones of 513 MB each, or 8 zones of 1,026 MB, satisfy this requirement. This flexibility enables smaller zones to reduce GC-induced tail latency while preserving NoWA's guarantee of SSD WAF = 1.

NoWA is worthwhile, even if it is imperfect. NoWA may not always completely eliminate SSD WA, since SSDs may reorder or relocate data depending on scheduling, load, flash chip state, open-superblock limits, or wear-leveling [66]. However, it still provides substantial practical benefit: even partial mitigation of page mixing reduces WA. Nevertheless, NoWA achieves SSD WAF = 1 on six enterprise SSDs from diverse vendors and capacities (Figure 14).

5.5 Using FDP Placement Hints

FDP exposes RU handles for host-side placement hints. When an SSD supports FDP, the NoWA pattern becomes unnecessary to achieve an SSD WAF of 1. In addition to the RU size, FDP-enabled SSDs expose *Reclaim Unit Handles (RUHs)*, each managing a disjoint set of RUs [76]. The number of RUHs determines how many independent write streams the SSD can maintain without multiplexing. The host can leverage this by providing placement hints when issuing writes, preventing pages with different deathtimes from being mixed internally [97]. In our design, this directly corresponds to the maximum number of active zones.

Optimization: using FDP to avoid multiplexing. FDP placement hints are straightforward to use in our design. We assign each zone a placement ID (PlID) modulo the number of RUHs and persist this mapping per zone. For example, with four RUHs (PlIDs 0–3), zones A–D map to PlIDs 0–3, so writes to zone A always use PlID 0. Under the write pattern shown in Figure 8, writes to different zones are directed to different RUHs, preventing interleaving between zones such as A and C (Figure 11). With the internal SSD information exposed by FDP, an SSD WAF of 1 is achievable as long as the zone size equals the RU size and the number of open zones does not exceed the RUH count (as shown in Table 3).

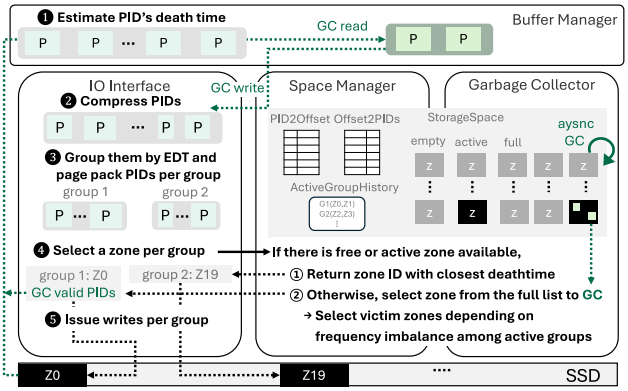


Figure 12: Overview of the write path with all optimizations

6 IMPLEMENTATION

From in-place to out-of-place writes. To evaluate our techniques, we extend LeanStore [63], a B-tree system that originally performs in-place writes. We integrate vmcache [64] as the buffer pool to support 1:N PID-offset mappings, enabling out-of-place writes. The resulting prototype, *Z (Zoned) LeanStore*, shows that our optimizations apply to an arbitrary page-based system. We first outline the write path to illustrate how the optimizations work together. We then describe necessary extensions to other components [31, 108] to implement our techniques without compromising durability.

6.1 Design Overview

How the main components support optimizations. We modify four DBMS components: the buffer manager, I/O interface, space manager, and garbage collector. The buffer manager computes each page's EDT using write timestamps stored in the page header. The I/O interface performs compression, page packing, and selects the backend based on the SSD type (e.g., ZNS or FDP). Storage space is partitioned into zones, sized according to the inferred SSD GC unit. The space manager maintains PID-offset and reverse mappings, tracks zone metadata, and selects zones for placement based on GDT. The GC reclaims space in both foreground and background, choosing victims based on GDT while preserving the NoWA pattern.

Write path. Write requests follow the path illustrated in Figure 12. Suppose a batch of PIDs must be flushed upon eviction. First, each PID's EDT is computed from previous write LSNs stored in the page header. The PID, its contents, and its EDT are passed to the I/O interface (❶), which compresses pages (❷) and groups PIDs with similar EDTs (❸). The space manager then assigns a target zone for each group (❹): if active zones exist, the one whose average EDT is closest is chosen (①); otherwise, a new zone is opened from the empty or full list, respecting the active-zone limit. If a zone is full, GC is triggered (②). During GC, valid pages not in the buffer pool are read in and rewritten following the same write path (❶-❸). When multiple zones are reclaimed, valid PIDs are sorted by descending EDT and written into zones with the highest invalidation counts, placing the coldest pages into the coldest zones. After GC completes, the originally requested write proceeds (❺).

6.2 Implementation Details

Sharing the buffer pool as a cache for GC. The GC shares the buffer pool with worker threads to perform GC I/Os, avoiding memory partitioning and allowing full utilization when GC is inactive. This reduces disk reads during GC as some valid PIDs may already be in memory. Although GC may harm the hit ratio, it often acts as a prefetcher, as GDT-based GC tends to load PIDs that will be accessed soon, thus reducing its negative impact.

Securing clean buffer frames for GC reads. A full buffer pool can create contention between eviction and GC, as both require frames. To avoid eviction writes during GC, only clean frames may be reused for GC reads. Thus, the system maintains a reserve of clean pages, integrated with fuzzy checkpointing [38]. If the reserve falls below the threshold, the checkpointer flushes pages during non-GC phases. In practice, this mechanism is rarely invoked because checkpointing already keeps enough clean frames available.

I/O buffer for compression and page packing. The I/O interface uses a per-thread buffer to issue I/O requests. PIDs are compressed with LZ4 [71], grouped by GDT, page-packed, and issued as an I/O request. For reads, a 4 KiB-aligned buffer fetches the data; after decompression, the requested PID is inserted into the buffer pool.

Space metadata. The space manager maintains three types of metadata and updates them whenever a write occurs. First, `PID2OffsetTable` maps each PID to its on-disk offset and compressed size. Second, `StorageSpace` provides the reverse mapping. It tracks which PIDs occupy each 4 KiB offset and stores zone metadata such as fill factor, page and zone invalidation counts, average EDT, and state. Finally, to implement the NoWA pattern, `ActiveGroupHistory` records the active group each time a new set of zones is opened.

Supporting various GC strategies. The garbage collector activates whenever active zones nearly run out of free device space. Depending on the enabled configuration, it runs victim selection strategies such as greedy, GDT-based GC, and/or NoWA-aware GC. Thus, the user can run GC that fits the workload characteristics.

Compatibility with ZNS SSDs and FDP. The I/O layer detects the device type and selects the appropriate backend via `ioctl`. Standard SSDs use an `io_uring`-based backend. For ZNS SSDs, writes use zone-append operations, zones are reset after GC, and the system tracks the device's limit on open zones. For FDP-enabled SSDs, we query the RU size and number of RU handles, set the maximum number of active zones accordingly, and apply placement hints by mapping zone IDs modulo the number of handles.

Logging and recovery. We use per-thread logs with continuous checkpointing [38]. To ensure recoverability, the system logs `PID2Offset` updates. We enforce write-ahead rules: page data is persisted before its mapping update is committed. Each checkpoint persists snapshots of the `PID2OffsetTable` and the `ActiveGroupHistory`. Recovery is performed by first locating the checkpoint LSN, reloading these snapshots, and replaying subsequent WAL entries; this naturally also reconstructs metadata. Finally, the `StorageSpace`layout is rebuilt from the final `PID2OffsetTable`.

7 EVALUATION

Our evaluation shows that the proposed out-of-place optimizations substantially reduce total WAF while improving throughput and SSD lifespan across various devices (with larger gains on ZNS or FDP), dataset sizes, and benchmarks. We focus primarily on write-related metrics, as minimizing flash writes is our main goal.

7.1 Setup and Methodology

Hardware setup. We evaluate on eight enterprise SSDs from five vendors. Experiments run on two servers: one with 360 GB DRAM and a 96-core AMD EPYC 9654P, and another with 500 GB DRAM and a 64-core Intel Xeon Gold 6342, both running Ubuntu 24.04 LTS. The first is used for general tests with 64 worker threads; the second for FDP-enabled SSDs with 32 worker threads.

OLTP benchmarks and DBMS configuration. We use two OLTP benchmarks. YCSB-A [16] (50% reads, 50% updates on a fixed-size dataset) is used for performance analysis, and TPC-C is used to assess generality and behavior under growing workloads. Both benchmarks target write-intensive OLTP scenarios that become I/O-bound once the dataset exceeds memory. To reflect realistic OLTP deployments, we set the buffer pool to 5–20% of the dataset to cache the effective working set [32, 33, 58]. The WAL size matches the buffer pool unless stated otherwise.

Evaluation method. Before each run, we issue `blkdiscard` to reset SSD mapping state. Benchmarks run until cumulative writes reach at least 4× the device capacity to ensure steady state [35]. We report average throughput, DB WAF, SSD WAF, buffer hit ratio, and CPU/memory usage over the final hour, after configurations have issued comparable writes. DB WAF is total DB-issued writes divided by eviction or checkpoint writes, and SSD WAF is total physical writes (via OCP [99], when supported) divided by DB-issued writes.

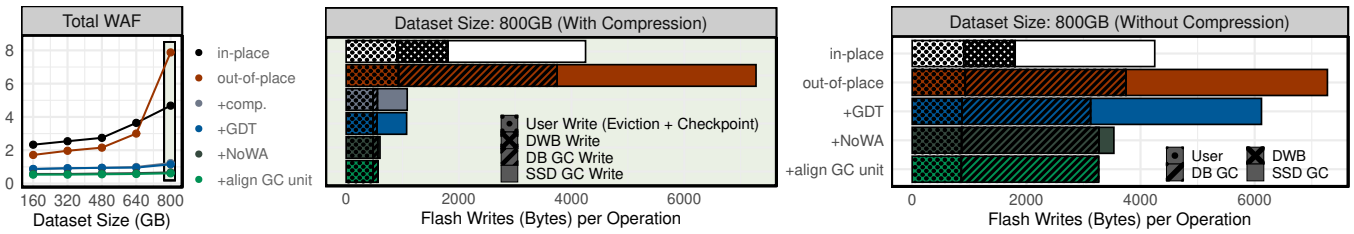
7.2 Main Performance Result

To analyze the impact of each optimization on total WAF, we run YCSB-A (skew = 0.8) across five dataset sizes on a Samsung PM9A3 SSD (894 GB), incrementally enabling our optimizations. Before introducing GDT, random placement and greedy GC are used. When out-of-place writes are enabled, up to 16 open zones are allowed. The zone size is set to 250 MB prior to aligning the GC unit and to 512 MB afterward, following the configuration inferred from the ZNS-like workload in Section 5.3 (16 × 512 MiB = 8 GB in total).

Total WAF. As shown in Figure 13a, each optimization reduces total WAF across all dataset sizes. The 160 GB dataset sees a 4.4× reduction (2.33→0.53), while the 800 GB dataset reaches a 7.8× reduction (4.72→0.60) with all optimizations enabled. Compression contributes the largest improvement by sharply lowering DB WAF. The NoWA pattern provides the second-largest reduction, with its benefits increasing for larger datasets.

Write drilldown (with compression). Figure 13b breaks down writes for the 800 GB dataset into user, DWB, DB GC, and SSD GC writes. In the in-place baseline, DWB dominates, accounting for half of all DB-issued writes. Switching to out-of-place writes removes DWB but makes DB GC the primary overhead, even increasing total WAF by 1.66×. Compression with GDT reduces DB GC writes, though SSD GC remains. Finally, the NoWA pattern together with aligned GC units eliminates SSD GC entirely, leaving user writes dominant and driving total WAF near the compression ratio.

Write drilldown (without compression). Compression shrinks the 800 GB dataset to 418 GB, expanding OP space to 53% of the SSD and reducing the GC valid ratio from 75% to 14% (Figure 13b). This



(a) Total WAF **(b) Write drilldown with compression** **(c) Write drilldown without compression**

Figure 13: Total WAF on different dataset sizes and write drilldown with 800 GB dataset (YCSB-A zipf θ = 0.8, Samsung PM9A3)

Table 1: Performance drilldown per proposed optimization (YCSB-A zipf θ = 0.8, DB size: 800 GB, Samsung PM9A3)

Version	OPS (K)	DB WAF	SSD WAF	Logical W. (Byte/op.)	Physical W. (Byte/op.)	Hit Ratio
in-place	229	2.00	2.36	1,858	4,378	0.932
→ out-of-place	230	4.06	1.94	3,745	7,274	0.918
+ comp. + pagepack	380	0.62	1.95	566	566	0.929
+ GDT	458	0.59	1.96	566	1,110	0.931
+ NoWA	510	0.60	1.07	567	606	0.931
+ aligning GC Unit	535	0.60	1.00	567	567	0.931
− comp.	328	3.58	1.00	3,364	3,364	0.927

lowers DB WAF and masks other effects, so we omit compression and page packing to isolate the remaining techniques in order of application. As shown in Figure 13c, out-of-place writes with all optimizations improve total WAF by 31% over in-place and 2.18× over naïve out-of-place. GDT reduces DB WAF from 4.02 to 3.36, while NoWA increases it to 3.58 due to compensation writes; the resulting drop in SSD WAF outweighs this increase. GDT mitigates DB WAF, while SSD-side optimizations (NoWA and aligning the GC unit) become more impactful without compression. Figure 15 shows that total WAF improves by at least 2× for smaller datasets.

Compression remains highly beneficial. Although the other optimizations are effective, compression remains highly beneficial and complements them well. Compression provides additional OP space for DB GC, lowering valid ratios by widening the invalidation window [35]. It also improves deathtime estimation by allowing more writes before GC triggers, giving a larger timestamp window. Together, these effects reduce DB WAF during GC. Compression also reduces space amplification and often improves throughput, making it a recommended choice when datasets are compressible.

7.3 Performance Drilldown

To examine how optimizations affect other metrics, Table 1 summarizes the 800 GB experiments in Figure 13.

Throughput. Applying all optimizations reduces total WAF and raises throughput from 229K OPS (in-place writes) to 535K. Fewer logical writes reduce time spent waiting for I/O [57, 106], and lower SSD WAF shortens I/O latency [35], further improving throughput. Compression with page packing adds further gains by reducing read traffic per operation.

DB WAF. For in-place, DB WAF is 2.00 due to double-write buffering. Out-of-place writes remove this cost, but with the dataset occupying 90% of SSD capacity and greedy DB GC, DB WAF rises to 4.06. Compression lowers DB WAF to 0.62 by both giving DB GC additional OP space and reducing the user write volume itself. GDT-based placement reduces it further to 0.59 by leveraging skew at GC time. The NoWA pattern slightly increases DB WAF to 0.60 due to compensation writes, but, as shown next, the reduction in SSD WAF outweighs this cost.

SSD WAF. In-place writes exhibit SSD WAF = 2.36, a value that varies across models (see Section 7.4). Switching to append-per-zone out-of-place writes reduces this to 1.94, likely because the large sequential write pattern aligns better with SSD internals [77]. The NoWA pattern further reduces SSD WAF to 1.07. While the effect of applying NoWA or GC-unit alignment alone varies by runtime, device model, and workload skew, applying both together consistently eliminates SSD WA, achieving WAF 1.0.

Logical & physical writes per operation. Columns 5–6 of Table 1 show logical (DB-issued) and physical (SSD-internal) bytes written per operation. With all optimizations, logical writes drop by 3.27× mainly due to compression; without compression they increase to 1.81×. Physical writes consistently decrease as SSD WAF approaches 1, eventually converging to logical writes. Overall, the techniques substantially reduce physically executed writes.

Buffer hit ratio. GC I/O can reduce buffer hit ratios because GC reads may evict cached pages. The largest drop occurs when switching from in-place to out-of-place writes (93.2%→91.8%). Compression improves the valid ratio and restores the hit ratio to 92.9%; GDT increases it slightly to 93.1%, since pages read during GC are more likely to be accessed soon (effectively acting as prefetching). Overall, the optimizations have minimal impact on hit ratio.

CPU overhead. In-place writes use 5% CPU, while out-of-place writes with all optimizations use 8.3%. Most overhead comes from securing buffer frames for GC reads. Since in-place writes already waste 8% CPU waiting for synchronous DWB, this extra usage is largely hidden. Future work will reduce overhead by improving hash-table scans when locating clean frames.

Memory usage. We kept the buffer pool size the same across in-place and out-of-place versions to match user writes per operation. The out-of-place schemes require extra metadata (Section 6), which is kept in memory, amounting to up to 10.9 GB in the worst case when the device is fully utilized. This bound varies with system configuration, such as compression ratio and zone size. If memory is constrained, the system can keep only active metadata and spill the rest to disk, which we leave to future work.

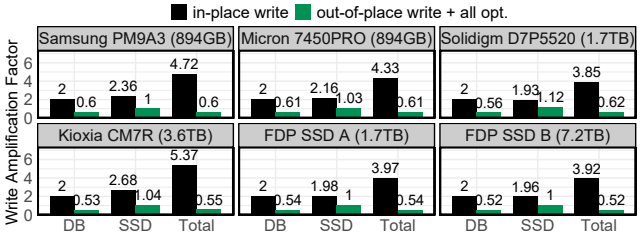


Figure 14: WAF comparison across different SSD models (YCSB-A zipf θ = 0.8, SSD space 90% full)

Table 2: Throughput comparison on CNS and ZNS SSDs (YCSB-A zipf θ = 0.8; capacities: CNS = 894 GB, ZNS = 951 GB)

	Configuration	Used SSD	Logical DB Size (GB)	OPS (K)
in-place vs. out-of-place	in-place	CNS	800	156.8
	oop + all opt.	CNS	800	322.2
	oop + comp. + GDT	ZNS	800	330.3
CNS vs. ZNS larger DB size	oop + all opt.	CNS	1,500	250.7
	oop + comp. + GDT	ZNS	1,500	328.9
	oop + comp. + GDT	ZNS	1,596	277.5

7.4 Results Across Various Commodity SSDs

Different commodity SSDs. To validate our optimizations across SSD models, we repeat the experiment on five additional enterprise SSDs, scaling datasets to 90% of each device's capacity (YCSB-A, zipf θ=0.8, 80 GB buffer pool). Figure 14 reports DB, SSD, and total WAF. Out-of-place writes consistently reduce total WAF over in-place, with improvements ranging from 6.2× (Solidigm D7-P5520) to 9.76× (Kioxia CM7-R), demonstrating robustness across models, vendors, and capacities. Larger SSDs exhibit lower DB WAF due to having more OP space (e.g., 7.2 TB → 752 GB vs. 894 GB → 94 GB). SSD WAF under out-of-place remains near the optimal value of 1 for all devices except the Solidigm D7-P5520 (1.12), which inherently shows slightly higher WAF even on sequential workloads [35]. In contrast, SSD WAF under in-place ranges from 1.93 to 2.68. Overall, the optimizations reliably minimize SSD WAF across devices.

7.5 ZNS and FDP

Out-of-place writes benefit CNS, but even more so on ZNS. ZNS SSDs offer two key advantages over standard SSDs: (1) they naturally maintain SSD WAF = 1 without requiring NoWA, and (2) they expose more usable capacity because no internal OP is needed. To evaluate these advantages, we run YCSB-A (zipf θ = 0.8) on a 951 GB ZNS SSD and an 894 GB CNS SSD with identical firmware. Since CNS cannot report physical writes (no OCP support), we compare throughput instead. With an 800 GB dataset (first group in Table 2), CNS with out-of-place writes doubles the throughput of in-place writes, and ZNS further improves throughput—2.1× over CNS in-place and 1.03× over CNS out-of-place.

ZNS vs. CNS with the same dataset size. To isolate the benefit of increased usable capacity on ZNS, we evaluate both devices with a 1,500 GB dataset (second group in Table 2). CNS in-place is omitted since the dataset exceeds its usable capacity. ZNS outperforms CNS by 31%, primarily due to lower DB WAF: the additional space reduces the valid ratio during DB GC and avoids NoWA-induced compensation writes.

ZNS vs. CNS at the same fill factor. We also compare the devices at the same *relative* fill. Since 1,500 GB on an 894 GB CNS matches the fill factor of 1,596 GB on a 951 GB ZNS, we evaluate these configurations (final row in Table 2). Here, ZNS outperforms CNS by only ~10%, much smaller than the previous 31% gain. This indicates that most of the benefit stems from ZNS's larger usable capacity, while removing NoWA overhead yields a smaller incremental gain.

SSD WAF = 1 with NoWA vs. FDP. To isolate the effect of FDP, we evaluate three configurations: (1) in-place writes; (2) out-of-place writes with all optimizations, using NoWA to ensure SSD WAF = 1; and (3) out-of-place writes with all optimizations except NoWA, using FDP placement hints to ensure SSD WAF = 1 instead. Table 3 reports DB WAF, SSD WAF, total WAF, and throughput for these configurations on YCSB-A (zipf θ = 0.8) using a 1.6 TB dataset. Both out-of-place configurations achieve SSD WAF = 1, whereas in-place writes show SSD WAF = 1.98. Without FDP, sustaining SSD WAF = 1 requires the NoWA pattern, whose compensation writes raise DB WAF to 0.57. With FDP enabled, placement hints prevent multiplexing directly and further reduce DB WAF to 0.54. As a result, total WAF decreases and throughput improves correspondingly.

Table 3: Effect of FDP on WAFs and throughput (YCSB-A zipf θ=0.8, 1.6 TB dataset, FDP SSD A)

Configuration	FDP Enabled?	DB WAF	SSD WAF	Total WAF	OPS (K)
in-place	disabled	2.00	1.98	3.96	437
oop + all opt.	disabled	0.57	1.00	0.57	541
oop + all opt. – NoWA	enabled	0.54	1.00	0.54	553

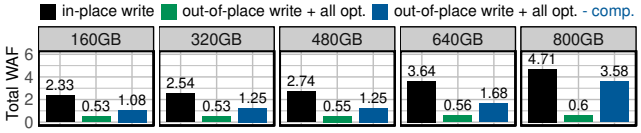


Figure 15: Total WAF while varying dataset size (YCSB-A zipf θ = 0.8, Samsung PM9A3)

7.6 Effects of Workload Variation

Varying dataset size. We run YCSB-A (zipfian θ = 0.8) with dataset sizes from 160 GB to 800 GB, setting the buffer pool to 10% of each dataset. Figure 15 reports total WAF for in-place writes, out-of-place writes with all optimizations, and out-of-place writes without compression. With in-place writes, total WAF increases from 2.33 (160 GB) to 4.72 (800 GB), whereas out-of-place writes with all optimizations remain low (0.53-0.60). Without compression, WAF rises from 1.08 to 3.58 but still stays well below in-place writes. The benefits of our optimizations become more pronounced as datasets grow: in-place writes incur rising SSD WAF, while out-of-place writes maintain SSD WAF = 1. Overall, our approach reduces both SSD and DB amplification, even without compression.

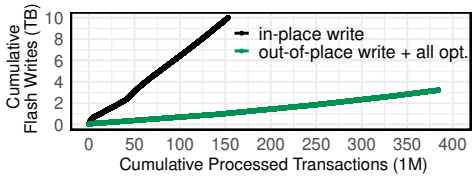


Figure 16: Comparison of cumulative flash writes per transaction during the same runtime (TPC-C 1.6 TB, FDP SSD A)

Evaluating with TPC-C. Beyond YCSB-A, we run TPC-C [102], whose dataset grows as new orders arrive (unlike YCSB-A's fixed size). We load 15,000 warehouses on FDP SSD A and run two-hour tests with in-place writes and with optimized out-of-place writes. Figure 16 plots cumulative flash writes versus transactions. In the same runtime, the optimized out-of-place configuration completes 2.45× more new-order transactions; for the same transaction count, in-place writes incur 7.2× more flash writes. These results show our approach remains effective under a different workload.

8 RELATED WORK

Different ways to perform out-of-place writes. Out-of-place writes appear in several forms. WiredTiger [109] stores each table in a cyclic file and rewrites valid pages to the file head during compaction. LSM engines [26, 30, 100] also write out-of-place, reclaiming space only when compaction is triggered, making GC timing and reclaimed space hard to control. In contrast, ZLeanStore reclaims space proactively at zone granularity [1, 13, 44, 52, 60, 91], independent of other DBMS components [70]. Doing so at the DBMS layer allows GC to account for both DB- and SSD-level WA.

Optimizing data structures to reduce DB WA. A substantial amount of work has reduced WA by optimizing index structures. B-tree-based storage engines such as WO-B-tree [31], Bf-tree [37], and B-epsilon tree [11] buffer updates and apply them in batches to reduce index-induced WA. LSM-tree engines were introduced to address this, and various works attempt to optimize it further [18, 19, 22, 26, 30, 51, 68, 88, 92, 104]. These approaches are orthogonal to ours: they reduce WA at the index level, whereas our techniques reduce DB WA at the page level without changing the index structure. Thus, they can be complementary when applied together.

Reducing WA by mitigating space amplification. Write volume can be reduced by lowering overall storage consumption. Techniques such as XMerge [3] and B-tree node compaction [56] improve node utilization and reduce disk writes; these index-level optimizations are orthogonal and can benefit from our approach. Disk-level compression in MySQL [79], WiredTiger [109], and RocksDB [26] similarly reduces write volume and storage footprint, but is ineffective under 4 KiB page granularity, whereas we provide a solution for 4 KiB pages. Moreover, out-of-place–write engines do not examine how compression interacts with DB WAF induced by GC, whereas we explicitly analyze and address these effects. Some SSDs also employ internal compression to increase overprovisioning and mitigate WA [122]; our DB GC naturally benefits from compression as well, but, unlike SSDs, we can flexibly enable, disable, or tune compression based on application needs.

Mitigating WA upon GC. Prior work proposes a wide range of GC algorithms that classify pages by lifetime, recency, or spatial locality to reduce WA during GC [20, 24, 42, 54, 60, 69, 70, 77, 94, 95, 116, 120]. Our approach differs in that we predict deathtimes of individual pages using database-specific semantics, which lower layers such as SSD GC cannot exploit [40, 54, 55]. To our knowledge, this is the first work to integrate GDT-based GC within a DBMS, leveraging workload knowledge that is only available to the DBMS.

Optimizing system write patterns for SSDs. Prior work has examined how applications can tailor I/O to SSD characteristics to exploit internal parallelism [39, 40, 47], analyzed tree-based access patterns [25], and characterized SSD behavior under diverse workloads [111, 112]. Although these studies improve latency or throughput, they do not explicitly target SSD WAF; even when WAF is considered, it is not fully eliminated [17, 115]. In contrast, our DBMS GC directly minimizes SSD WAF while accounting for its interaction with DB WAF to reduce total WAF.

Zoned storage for ZNS. Several systems [13, 34, 44, 61, 98, 114] leverage ZNS SSDs [12] to exploit the SSD WAF of 1. While effective, these designs are tied to ZNS hardware and thus limited to environments where such devices are available. In contrast, our approach treats ZNS as an optional optimization rather than a requirement, making it applicable across a broader range of devices.

Systems using FDP. The FDP interface was introduced only recently [75], and emerging work explores its potential for reducing WAF [4, 29, 118]. Our work identifies the optimal use case for FDP placement hints from the perspective of out-of-place DBMSs to achieve an SSD WAF of 1. We further suggest appropriate settings for DB GC granularity and the number of active zones under FDP.

9 CONCLUSION

This paper demonstrates that co-designing a DBMS and an SSD can improve throughput and extend SSD lifespan. Our four out-of-place schemes reshape the DBMS write pattern to mitigate WA at both the DB and SSD layers. We further show that the design naturally leverages ZNS and FDP. Future work includes extending these techniques to LSM engines, supporting multi- or shared-device deployments, and exploring disk- and HM-SMR–based systems.

Beyond DBMSs and SSDs, our findings apply broadly to storage systems. Our large sequential write patterns also benefit disk-based systems [1, 14, 67]. Systems with their own translation layer and garbage collection can adopt our DB-level WAF optimizations, while log-structured filesystems can use the NoWA pattern to reduce aging [46, 84]. We hope this work helps storage designers better understand the implications of system writes on devices and how to fully exploit device capabilities.

ACKNOWLEDGMENTS

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 551650645. The authors thank the Samsung Memory Research Center (SMRC), Yongho Song, and Meongchul Song for providing the infrastructure used for FDP experiments. We acknowledge Western Digital and Matias Bjørling for providing the ZNS and comparison SSDs. We thank the reviewers for their insightful questions. Selected responses are included in the appendix of the extended version [59].

REFERENCES

[1] Abutalib Aghayev and Peter Desnoyers. 2015. Skylight—A Window on Shingled Disk Operation. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 135–149. https://www.usenix.org/conference/fast15/technical-sessions/presentation/aghayev

[2] Abutalib Aghayev, Sage A. Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2020. The Case for Custom Storage Backends in Distributed Storage Systems. *ACM Trans. Storage* 16, 2 (2020), 9:1–9:31.

[3] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*. www.cidrdb.org, Virtual Event, 8. https://vldb.org/cidrdb/2021/contention-and-space-management-in-b-trees.html

[4] Michael Allison, Arun George, Javier González, Dan Helmick, Vikash Kumar, Roshan R. Nair, and Vivek Shah. 2025. Towards Efficient Flash Caches with Emerging NVMe Flexible Data Placement SSDs. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, March 30 - April 3, 2025*. ACM, Rotterdam, The Netherlands, 1142–1160. https://doi.org/10.1145/3689031.3696082

[5] Mijin An, Soojun Im, Dawoon Jung, and Sang-Won Lee. 2022. Your Read is Our Priority in Flash Storage. *Proc. VLDB Endow.* 15, 9 (2022), 1911–1923. https://www.vldb.org/pvldb/vol15/p1911-lee.pdf

[6] Mijin An, In-Yeong Song, Yong-Ho Song, and Sang-Won Lee. 2022. Avoiding Read Stalls on Flash Storage. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1404–1417. https://doi.org/10.1145/3514221.3526126

[7] Remzi H. Arpaci-Dusseau. 2017. Operating Systems: Three Easy Pieces. *login Usenix Mag.* 42, 1 (2017), 4. https://www.usenix.org/publications/login/spring2017/arpaci-dusseau

[8] Jens Axboe. 2026. fio: Flexible I/O Tester. https://github.com/axboe/fio. Accessed: 2026-03-11.

[9] Brenda S. Baker. 1985. A New Proof for the First-Fit Decreasing Bin-Packing Algorithm. *J. Algorithms* 6, 1 (1985), 49–70.

[10] Andy Banta and Scott Shadley. 2025. Total Cost and Performance of SSDs. SNIA Storage Developer Conference (SDC). https://youtu.be/XsfnItj05-E. Accessed: 2025-11-08.

[11] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bε-trees and Write-Optimization. *login Usenix Mag.* 40, 5 (2015), 7. https://www.usenix.org/publications/login/oct15/bender

[12] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *USENIX ATC*. USENIX Association, Virtual Event, 689–703.

[13] Matias Bjørling et al. 2021. ZenFS: A RocksDB File System for Zoned Name Spaces. https://github.com/westerndigitalcorporation/zenfs. Accessed: 2025-08-03.

[14] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *SOSP*. ACM, Virtual Event / Koblenz, Germany, 836–850.

[15] Chandranil Chakraborttii and Heiner Litz. 2021. Reducing Write Amplification in Flash by Death-Time Prediction of Logical Block Addresses. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '21)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages. https://doi.org/10.1145/3456727.3463784

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, Indianapolis, Indiana, USA, 143–154.

[17] Matthew Curtis-Maury, Ram Kesavan, Bharadwaj V. R., Nikhil Mattankot, Vania Fang, Yash Trivedi, Kesari Mishra, and Qin Li. 2024. I/O in a Flash: Evolution of ONTAP to Low-Latency SSDs. In *FAST*. USENIX Association, Santa Clara, CA, USA, 177–192.

[18] Niv Dayan and Manos Athanassoulis. 2018. Monkey: Optimal Tuning of LSM-Tree Compaction and Bloom Filters. *ACM Trans. Database Syst.* 43, 4 (2018), 15:1–15:48. https://doi.org/10.1145/3236190

[19] Niv Dayan and Manos Athanassoulis. 2022. Hybrid Compaction: Adaptive LSM-Tree Compaction to Improve Write Amplification. *Proc. VLDB Endow.* 15, 12 (2022), 3071–3084. https://vldb.org/pvldb/vol15/p3071-dayan.pdf

[20] Niv Dayan and Philippe Bonnet. 2015. Garbage Collection Techniques for Flash-Resident Page-Mapping FTLs. arXiv:1504.01666. arXiv:1504.01666 https://arxiv.org/abs/1504.01666 Accessed: 2026-03-11.

[21] Niv Dayan, Luc Bouganim, and Philippe Bonnet. 2015. Modelling and Managing SSD Write-amplification. arXiv:1504.00229. arXiv:1504.00229 [cs.DB] https://arxiv.org/abs/1504.00229 Accessed: 2026-03-11.

[22] Niv Dayan, Valentin Nathan, and Stratos Idreos. 2024. How to Grow an LSM Tree. *Proc. ACM Manag. Data* 2, 2 (2024), 1–27. https://doi.org/10.1145/3725310

[23] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. 2022. Spooky: granulating LSM-tree compactions correctly. *Proc. VLDB Endow.* 15, 11 (July 2022), 3071–3084. https://doi.org/10.14778/3551793.3551853

[24] Peter Desnoyers. 2014. Analytic Models of SSD Write Performance. *ACM Trans. Storage* 10, 2 (2014), 8:1–8:25.

[25] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. 2020. Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs. *Proc. VLDB Endow.* 14, 3 (2020), 364–377.

[26] Siying Dong, Mark D. Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB. In *Conference on Innovative Data Systems Research*. www.cidrdb.org, Chaminade, CA, USA, 9. https://www.cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf Accessed: 2026-03-11.

[27] facebook. 2024. Zstandard - Fast real-time compression algorithm. https://github.com/facebook/zstd. Accessed: 2026-03-11.

[28] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (2022), 3535–3547.

[29] Arun George et al. 2024. FDP Integration in CacheLib - the Future of Memory and Storage. https://files.futurememorystorage.com/proceedings/2024/20240808_OPSW-302-1_George.pdf. Accessed: 2025-08-03.

[30] google. 2021. leveldb. https://github.com/google/leveldb. Accessed: 2026-03-11.

[31] Goetz Graefe. 2004. Write-Optimized B-Trees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3, 2004*. Morgan Kaufmann, Toronto, Canada, 672–683. https://doi.org/10.1016/B978-012088469-8.50060-7

[32] Jim Gray. 1991. The DebitCredit Transaction Benchmark. *Datamation* 37, 7 (1991), 59–64.

[33] Jim Gray and Franco Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. ACM, San Francisco, CA, USA, 395–398. https://doi.org/10.1145/38713.38755

[34] Jinyong Ha and Yongseok Son. 2025. zCeph: Design and implementation of a ZNS-friendly distributed file system. *Future Gener. Comput. Syst.* 169 (2025), 107763.

[35] Gabriel Haas, Bohyun Lee, Philippe Bonnet, and Viktor Leis. 2025. SSD-iq: Uncovering the Hidden Side of SSD Performance. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4295–4308. https://doi.org/10.14778/3749646.3749694

[36] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (2023), 2090–2102.

[37] Xiangpeng Hao and Badrish Chandramouli. 2024. Bf-Tree: A Modern Read-Write-Optimized Concurrent Larger-Than-Memory Range Index. *Proc. VLDB Endow.* 17, 11 (2024), 3442–3455.

[38] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 877–892. https://doi.org/10.1145/3318464.3389716

[39] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. *Proc. VLDB Endow.* 16, 7 (2023), 1712–1725.

[40] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *EuroSys*. ACM, Belgrade, Serbia, 127–144.

[41] John Hines, Nathan Cunningham, and Galo H. Alférez. 2023. Performance Comparison of Operations in the File System and in Embedded Key-Value Databases. In *Intelligent Computing*, Kohei Arai (Ed.). Lecture Notes in Networks and Systems, Vol. 739. Springer, Cham, Switzerland, 386–400. https://doi.org/10.1007/978-3-031-37963-5_27

[42] Benny Van Houdt. 2014. On the necessity of hot and cold data identification to reduce the write amplification in flash-based SSDs. *Perform. Evaluation* 82 (2014), 1–14.

[43] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proc. VLDB Endow.* 17, 3 (Nov. 2023), 577–590. https://doi.org/10.14778/3632093.3632117

[44] Inhwi Hwang, Sangjin Lee, Sunggon Kim, Hyeonsang Eom, and Yongseok Son. 2025. Z-LFS: A Zoned Namespace-tailored Log-structured File System for Commodity Small-zone ZNS SSDs. In *USENIX ATC*. USENIX Association, Boston, MA, USA, 547–562.

[45] Soojun Im and Dongkun Shin. 2010. ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer. *J. Syst. Archit.* 56, 12 (2010), 641–653.

[46] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R. Ganger. 2018. Geriatrix: Aging what you see and what you don't see. A file system aging approach

for modern storage systems. In *USENIX ATC*. USENIX Association, Boston, MA, USA, 691–704.

[47] Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian Kroth. 2019. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. *PVLDB* 13, 4 (2019), 519–532.

[48] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-Streamed Solid-State Drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage '14, Philadelphia, PA, USA, June 18, 2014*. USENIX Association, Philadelphia, PA, USA, 5. https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang

[49] Minji Kang, Soyee Choi, Gihwan Oh, and Sang Won Lee. 2020. 2R: Efficiently Isolating Cold Pages in Flash Storages. *PVLDB* 13, 11 (2020), 2004–2017.

[50] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, Snowbird, UT, USA, June 22-27, 2014*. ACM, Snowbird, UT, USA, 529–540. https://doi.org/10.1145/2588555.2595632

[51] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for nonvolatile memory with NoveLSM. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 993–1005.

[52] Ryusuke Konishi, Yoshiji Amagai, Hisashi Hifumi, Seiji Kihara, Kazutaka Maeda, and Satoshi Moriai. 2006. The Linux Implementation of a Log-Structured File System. *SIGOPS Oper. Syst. Rev.* 40, 3 (2006), 102–107. https://doi.org/10.1145/1151374.1151380

[53] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. *Proc. ACM Manag. Data* 2, 6 (2024), 238:1–238:27.

[54] Tomer Lange, Joseph (Seffi) Naor, and Gala Yadgar. 2023. Offline and Online Algorithms for SSD Management. *Commun. ACM* 66, 7 (2023), 129–137.

[55] Tomer Lange, Joseph (Seffi) Naor, and Gala Yadgar. 2025. Optimal SSD Management with Predictions. *Proc. ACM Meas. Anal. Comput. Syst.* 9, 2 (2025), 1–28.

[56] Bo-Hyun Lee, Mijin An, and Sang-Won Lee. 2023. A Case for Space Compaction of B-Tree Nodes on Flash Storage. *IEEE Access* 11 (2023), 38149–38156.

[57] Bo-Hyun Lee, Mijin An, and Sang-Won Lee. 2023. LRU-C: Parallelizing Database I/Os for Flash SSDs. *Proc. VLDB Endow.* 16, 9 (2023), 2364–2376.

[58] Bohyun Lee, Seongjae Moon, Jonghyeok Park, and Sang-Won Lee. 2025. Boosting OLTP Performance with Per-Page Logging on NVDIMM. *Proc. ACM Manag. Data* 3, 1, Article 17 (Feb. 2025), 28 pages. https://doi.org/10.1145/3709667

[59] Bohyun Lee, Tobias Ziegler, and Viktor Leis. 2026. How to Write to SSDs. arXiv:2603.09927. arXiv:2603.09927 [cs.DB] https://arxiv.org/abs/2603.09927 Accessed: 2026-03-11.

[60] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *FAST*. USENIX Association, Santa Clara, CA, USA, 273–286.

[61] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In *HotStorage*. ACM, Virtual Event, 93–99.

[62] Sang-Won Lee and Bongki Moon. 2007. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD Conference*. ACM, Beijing, China, 55–66.

[63] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17, 12 (2024), 4536–4545.

[64] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1 (2023), 7:1–7:25.

[65] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD Conference*. ACM, Virtual Event, China, 2852–2858.

[66] Alberto Lerner and Philippe Bonnet. 2024. *Principles of Database and Solid-State Drive Co-Design*. Springer Cham, Cham, Switzerland. https://doi.org/10.1007/978-3-031-57877-9

[67] Jinhong Li, Qiuping Wang, and Patrick P. C. Lee. 2022. Efficient LSM-Tree Key-Value Data Management on Hybrid SSD/HDD Zoned Storage. arXiv:2205.11753. arXiv:2205.11753 https://arxiv.org/abs/2205.11753 Accessed: 2026-03-11.

[68] Junfeng Liu, Haoxuan Xie, and Siqiang Luo. 2025. ArceKV: Towards Workload-driven LSM-compactions for Key-Value Stores Under Dynamic Workloads. arXiv:2508.03565. arXiv:2508.03565 [cs.DB] https://arxiv.org/abs/2508.03565 Accessed: 2026-03-11.

[69] David Lomet and Chen Luo. 2020. Efficiently Reclaiming Space in a Log Structured Store. arXiv:2005.00044 [cs.DB] https://arxiv.org/abs/2005.00044

[70] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Trans. Storage* 13, 1 (2017), 5:1–5:28.

[71] lz4. 2024. LZ4 - Extremely fast compression. https://github.com/lz4/lz4. Accessed: 2026-03-11.

[72] Umesh Maheshwari. 2021. From blocks to rocks: a natural extension of zoned namespaces. In *HotStorage*. ACM / USENIX Association, Virtual Event, 21–27.

[73] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2022. Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *FAST*. USENIX Association, Santa Clara, CA, USA, 165–180.

[74] MariaDB Foundation. 2025. InnoDB Doublewrite Buffer. https://mariadb.com/kb/en/innodb-doublewrite-buffer/. Explains MariaDB's implementation of doublewrite buffering for crash recovery and data integrity. Accessed: 2026-03-11.

[75] Bill Martin, Judy Brock, Dan Helmick, Robert Moss, Mike Allison, Benjamin Lim, Jiwon Chang, Ross Stenfort, Young Ahn, Wei Zhang, et al. 2022. TP4146: Flexible Data Placement. https://nvmexpress.org/wp-content/uploads/NVM-Express-2.0-Ratified-TPs_20230111.zip. Ratified 2022-11-30. Retrieved 2025-08-03 Accessed: 2026-03-11.

[76] John Rudelic Michael Allison. 2023. What is the NVM Express® Flexible Data Placement (FDP)? https://www.snia.org/sniadeveloper/session/18524. Accessed: 2026-03-11.

[77] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: random write considered harmful in solid state drives. In *FAST*. USENIX Association, San Jose, CA, USA, 12.

[78] MySQL Team, Oracle Corp. 2022. Doublewrite Buffer. https://dev.mysql.com/doc/refman/8.4/en/innodb-doublewrite-buffer.html. Accessed: 2026-03-11.

[79] MySQL Team, Oracle Corp. 2022. InnoDB Page Compression. https://dev.mysql.com/doc/refman/8.4/en/innodb-page-compression.html. Accessed: 2026-03-11.

[80] Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2025. Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs. *Proc. ACM Manag. Data* 3, 3 (2025), 191:1–191:24.

[81] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, Utrecht, The Netherlands, 3878–3892. https://doi.org/10.1109/ICDE60146.2024.00297

[82] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. 2010. Workload-aware storage layout for database systems. In *SIGMOD Conference*. ACM, Indianapolis, Indiana, USA, 939–950.

[83] Jonghyeok Park, Soyee Choi, Gihwan Oh, Soojun Im, Moon-Wook Oh, and Sang-Won Lee. 2023. FlashAlloc: Dedicating Flash Blocks by Objects. *Proc. VLDB Endow.* 16, 11 (July 2023), 3266–3278. https://doi.org/10.14778/3611479.3611524

[84] Jonggyu Park and Young Ik Eom. 2022. File fragmentation from the perspective of I/O control. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022*, Ali Anwar, Dimitris Skourtis, Sudarsun Kannan, and Xiaosong Ma (Eds.). ACM, Virtual Event, 126–132. https://doi.org/10.1145/3538643.3539746

[85] Seon-Yeong Park, Dawoon Jung, Jeong-Uk Kang, Jin-Soo Kim, and Joonwon Lee. 2006. CFLRU: A Replacement Algorithm for Flash Memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (Seoul, Korea) *(CASES '06)*. Association for Computing Machinery, New York, NY, USA, 234–241. https://doi.org/10.1145/1176760.1176789

[86] Percona LLC. 2025. XtraDB Doublewrite Buffer. https://docs.percona.com/percona-server/8.0/encrypting-doublewrite-buffers.html?h=doublewrite+buffer. Documents the Percona XtraDB doublewrite mechanism inherited from InnoDB. Accessed: 2026-03-11.

[87] PostgreSQL Global Development Group. 2025. The Oversized-Attribute Storage Technique (TOAST). https://www.postgresql.org/docs/current/storage-toast.html. https://www.postgresql.org/docs/current/storage-toast.html Accessed: 2025-08-26.

[88] TigerBeetle Project. 2025. TigerBeetle: High-performance financial transactions database. https://github.com/tigerbeetle/tigerbeetle. Accessed: 2025-11-08.

[89] Devashish R. Purandare, Peter Alvaro, Avani Wildani, Darrell D. E. Long, and Ethan L. Miller. 2025. Valet: Efficient Data Placement on Modern SSDs. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '25)*. ACM, Online, USA, 1–14. https://doi.org/10.1145/3772052.3772256 arXiv:arXiv:2501.00977v2 [cs.OS]

[90] Varsha Rao and Andrew A. Chien. 2025. Understanding the Operational Carbon Footprint of Storage Reliability and Management. *SIGENERGY Energy Inform. Rev.* 4, 5 (April 2025), 180–187. https://doi.org/10.1145/3727200.3727227

[91] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, SOSP '91, Pacific Grove, CA, USA, October 13-16, 1991*. ACM, Pacific Grove, CA, USA, 1–15. https://doi.org/10.1145/121132.121137

[92] Subhadeep Sarkar, Dimitris Staratzis, Ziehen Zhu, and Manos Athanassoulis. 2021. Constructing and analyzing the LSM compaction design space. *Proc. VLDB Endow.* 14, 11 (July 2021), 2216–2229. https://doi.org/10.14778/3476249.3476274

[93] Russell Sears, Catharine van Ingen, and Jim Gray. 2007. To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? arXiv:cs/0701168. arXiv:cs/0701168 https://arxiv.org/abs/cs/0701168 Accessed: 2026-03-11.

[94] Mansour Shafaei, Peter Desnoyers, and Jim Fitzpatrick. 2016. Write Amplification Reduction in Flash-Based SSDs Through Extent-Based Temperature Identification. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*, Nitin Agrawal and Sam H. Noh (Eds.). USENIX Association, Denver, CO, USA, 5. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/shafaei

[95] Jingcheng Shen, Lang Yang, Linbo Long, Zhenhua Tan, Congming Gao, Kan Zhong, Masao Okita, and Fumihiko Ino. 2025. Overlapping Aware Data Placement Optimizations for LSM Tree-Based Store on ZNS SSDs. *ACM Trans. Archit. Code Optim.* 22, 2, Article 57 (June 2025), 25 pages. https://doi.org/10.1145/3721287

[96] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael J. Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: A DBMS-oriented Operating System. *Proc. VLDB Endow.* 15, 1 (2021), 21–30.

[97] Inho Song, Shoaib Asif Qazi, Javier González, Matias Bjørling, Sam H. Noh, and Huaicheng Li. 2026. Characterizing and Emulating FDP SSDs with WARP. In *Proceedings of the 24th USENIX Conference on File and Storage Technologies (FAST '26)*. USENIX Association, Santa Clara, CA, USA, 17. https://www.usenix.org/conference/fast26/presentation/song

[98] SSDFS Project. 2024. SSDFS: Current Status. Conference presentation at Linux Plumbers Conference 2024. https://lpc.events/event/18/contributions/1821/attachments/1387/2992/SSDFS-current-status-LPC-2024-22-07-2024.pptx.pdf Accessed: 2025-08-05.

[99] Ross Stenfort, Ta-Yu Wu, and Lee Prewitt. 2020. *NVMe Cloud SSD Specification*. Open Compute Project (OCP). Version 1.0a (06262020).

[100] TigerBeetle Systems. 2023. TigerBeetle Architecture Notes. https://github.com/tigerbeetle/tigerbeetle/. Accessed: 2026-03-11.

[101] The PostgreSQL Global Development Group. 2019. PostgreSQL documentation: non-durable settings (Section 14.5). https://www.postgresql.org/docs/current/runtime-config-wal.html#GUC-FULL-PAGE-WRITES. Accessed: 2026-03-11.

[102] Transaction Processing Performance Council (TPC). 2010. TPC BENCHMARK™ C Standard Specification. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. Accessed: 2026-03-11.

[103] Transactional Blog. 2025. Torn Write Detection and Protection in CedarDB. https://transactional.blog/blog/2025-torn-writes. Mentions CedarDB as another system using doublewrite buffering to protect from torn writes. Accessed: 2026-03-11.

[104] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. 2018. NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management. In *EDBT*. OpenProceedings.org, Vienna, Austria, 457–460.

[105] Demian E. Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *Proc. VLDB Endow.* 16, 11 (2023), 3323–3334.

[106] Hengrui Wang, Jiansheng Qiu, Fangzhou Yuan, and Huanchen Zhang. 2025. Rethinking The Compaction Policies in LSM-trees. *Proc. ACM Manag. Data* 3, 3 (2025), 207:1–207:26.

[107] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating Data via Block Invalidation Time Inference for Write Amplification Reduction in Log-Structured Storage. In *FAST*. USENIX Association, Santa Clara, CA, USA, 429–444.

[108] WiredTiger. 2020. Checkpoint Architecture. https://source.wiredtiger.com/mongodb-4.4/arch-checkpoint.html. https://source.wiredtiger.com/mongodb-4.4/arch-checkpoint.html Accessed: 2025-12-08.

[109] WiredTiger. 2023. Compaction. https://source.wiredtiger.com/develop/arch-compact.html. Accessed: 2026-03-11.

[110] Guanying Wu and Xubin He. 2012. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 10.

[111] Gala Yadgar and Moshe Gabel. 2016. Avoiding the Streetlight Effect: I/O Workload Analysis with SSDs in Mind. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, USA, June 20-21, 2016*, Nitin Agrawal and Sam H. Noh (Eds.). USENIX Association, Denver, CO, USA, 5. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/yadgar

[112] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. 2021. SSD-based Workload Characteristics and Their Performance Implications. *ACM Trans. Storage* 17, 1 (2021), 8:1–8:26.

[113] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. 2015. Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes. In *FAST*. USENIX Association, Santa Clara, CA, USA, 257–271.

[114] Chongzhuo Yang, Chang Guo, Ming Zhao, and Zhichao Cao. 2024. A Zoned Storage Optimized Flash Cache on ZNS SSDs. arXiv:2410.11260. arXiv:2410.11260 [cs.PF] https://arxiv.org/abs/2410.11260 Accessed: 2026-03-11.

[115] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't Stack Your Log On My Log. In *INFLOW*. USENIX Association, Broomfield, CO, USA, 10.

[116] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proc. VLDB Endow.* 16, 1 (2022), 99–112.

[117] Xiangqun Zhang, Janki Bhimani, Shuyi Pei, Eunji Lee, Sungjin Lee, Yoon Jae Seong, Eui Jin Kim, Changho Choi, Eyee Hyun Nam, Jongmoo Choi, and Bryan S. Kim. 2025. Storage Abstractions for SSDs: The Past, Present, and Future. *ACM Trans. Storage* 21, 1, Article 2 (Jan. 2025), 44 pages. https://doi.org/10.1145/3708992

[118] Xiangqun Zhang, Shuyi Pei, Jongmoo Choi, and Bryan S. Kim. 2023. Excessive SSD-Internal Parallelism Considered Harmful. In *Proceedings of the 15th ACM/USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2023, Boston, MA, USA, 9 July 2023*, Ali Anwar, Ningfang Mi, Vasily Tarasov, and Yiying Zhang (Eds.). ACM, Boston, MA, USA, 65–72. https://doi.org/10.1145/3599691.3603412

[119] Mai Zheng, Joseph A. Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDS under power fault. In *FAST*. USENIX, San Jose, CA, USA, 271–284.

[120] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. 2015. An efficient page-level FTL to optimize address translation in flash memory. In *EuroSys*. ACM, Bordeaux, France, 12:1–12:16.

[121] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine Using DRAM, NVMe, and RDMA. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 685–699. https://doi.org/10.1145/3514221.3526187

[122] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and How?. In *INFLOW*. USENIX Association, Broomfield, CO, USA, 10.