Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression

Yifan Qiao, *Rensselaer Polytechnic Institute;* Xubin Chen, Google Inc.;
Ning Zheng, Jiangpeng Li, and Yang Liu, *ScaleFlux Inc.;* Tong Zhang,
Rensselaer Polytechnic Institute and ScaleFlux Inc.

https://www.usenix.org/conference/fast22/presentation/qiao

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Closing the B$^+$-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression

Yifan Qiao[†]*, Xubin Chen*[‡]*, Ning Zheng*[*]*, Jiangpeng Li*[*]*, Yang Liu*[*]*, and Tong Zhang*[†*]

[†] *Rensselaer Polytechnic Institute, NY, USA*

[‡] *Google Inc., WA, USA* [*] *ScaleFlux Inc., CA, USA*

Abstract

This paper studies how B$^+$-tree could take full advantage of modern storage hardware with built-in transparent compression. Recent years witnessed significant interest in applying log-structured merge tree (LSM-tree) as an alternative to B$^+$-tree, driven by the widely accepted belief that LSM-tree has distinct advantages in terms of storage cost and write amplification. This paper aims to revisit this belief upon the arrival of storage hardware with built-in transparent compression. Advanced storage appliances and emerging computational storage drives perform hardware-based lossless data compression, transparent to OS and user applications. Beyond straightforwardly reducing the storage cost gap between B$^+$-tree and LSM-tree, such storage hardware creates new opportunities to re-think the implementation of B$^+$-tree. This paper presents three simple design techniques that can leverage such modern storage hardware to significantly reduce the B$^+$-tree write amplification. Experiments on a commercial storage drive with built-in transparent compression show that the proposed design techniques can reduce the B$^+$-tree write amplification by over 10×. Compared with RocksDB (a key-value store built upon LSM-tree), the enhanced B$^+$-tree implementation can achieve similar or even smaller write amplification.

1 Introduction

This paper investigates the implementation of B$^+$-tree upon a growing family of data storage hardware that internally carry out hardware-based lossless data compression, transparent to the host OS and user applications. Modern all-flash array products (e.g., Dell EMC PowerMAX [9], HPE Nimble Storage [14], and Pure Storage FlashBlade [28]) always come with the built-in hardware-based transparent compression capability. Commercial solid-state storage drives with built-in transparent compression are emerging (e.g., computational storage drive from ScaleFlux [31] and Nytro SSD from Seagate [13]). Moreover, Cloud vendors have started to integrate hardware-based compression capability into their storage infrastructure, e.g., Microsoft Corsia [7] and emerging DPU (data processing unit) [5], leading to imminent arrival of cloud-based storage hardware with built-in transparent compression. With dedicated hardware compression engines, such storage hardware support high-throughput data (de)compression at very low latency and zero host CPU overhead.

As the most widely used indexing data structure, B$^+$-tree [12] powers almost all the relational database management systems (RDBMs) today. Recently, log-structured merge tree (LSM-tree) [25] has attracted significant interest as a contender to B$^+$-tree, mainly because its data structure could enable better storage space usage efficiency and lower write amplification. The arrival of storage hardware with built-in transparent compression could straightforwardly reduce or even eliminate the storage cost gap between B$^+$-tree and LSM-tree. This paper shows that such storage hardware can also be leveraged to significantly reduce B$^+$-tree write amplification. The key is to exploit the fact that in-storage transparent compression allows data management software employ *sparse data structure* without sacrificing the true physical storage cost. When running on such storage hardware, data management software could leave 4KB LBA (logical block address) blocks partially filled or even completely empty, without wasting the physical storage space usage. Intuitively, the feasibility of employing sparse data structure creates a new spectrum of design space for innovating data management systems [36].

This paper shows that B$^+$-tree could employ sparse data structure enabled by in-storage transparent compression to largely reduce its write amplification. We note that write amplification is measured based on the amount of data being written to the physical storage media (i.e., after in-storage compression), other than the amount of data being written by the host (i.e., before in-storage compression). In particular, this paper presents three simple yet effective design techniques: (1) *deterministic page shadowing* that can ensure B$^+$-tree page update atomicity without incurring extra write overhead, (2) *localized page modification logging* that can reduce the write amplification caused by the mismatch between the B$^+$-tree page size and the size of data modification,

and (3) *sparse redo logging* that can reduce the write amplification caused by B$^+$-tree redo logging (or write-ahead logging). With significantly reduced write amplification, B$^+$-tree can support much higher insert/update throughput, and more readily accommodate low-cost, low-endurance NAND flash memory (e.g., QLC NAND flash memory).

Accordingly, we implemented a B$^+$-tree (called B$^-$-tree) that incorporates the three design techniques. We further compared it with LSM-tree (RocksDB [30]) and normal B$^+$-tree (WiredTiger [33]). We carried out experiments on a commercial computational storage drive with built-in transparent compression [31]. The results well demonstrate the effectiveness of the proposed design techniques on reducing the B$^+$-tree write amplification. For example, under random write workloads with 128B per record, RocksDB and WiredTiger (with page size of 8KB) have write amplification of 14 and 64, respectively, while our B$^-$-tree (with 8KB page size) has a write amplification of only 8, representing 43% and 88% reduction compared with RocksDB and WiredTiger, respectively. The smaller write amplification can directly translate into a higher write throughput. For example, our results show that, under random write workloads, B$^-$-tree can achieve about 85K TPS (transactions per second), while the TPS of RocksDB and WiredTiger is 71K and 28K, respectively. Moreover, we note that the proposed design techniques mainly confine within the I/O module of B$^+$-tree and are largely orthogonal to the other modules. Hence, it is relatively easy to incorporate these techniques into existing B$^+$-tree implementations. For example, upon a baseline B$^+$-tree implementation, we only modified/added about 1,200 LoC to realize the B$^+$-tree.

2 Background

2.1 B$^+$-tree Data Compression

B$^+$-tree manages its data storage in the unit of page. To reduce data storage cost, B$^+$-tree could apply block compression algorithms (e.g., lz4 [23], zlib [37], and ZSTD [38]) to compress each on-storage page (e.g., the page compression feature in MySQL and MongoDB/WiredTiger). In addition to the obvious CPU overhead, B$^+$-tree page compression suffers from compression ratio loss due to the 4KB-alignment constraint, which can be explained as follows: Modern storage devices serve I/O requests in the unit of 4KB LBA blocks. As a result, each B$^+$-tree page (regardless of compressed or uncompressed) must entirely occupy one or multiple 4kB LBA blocks on the storage device (i.e., no two pages could share one LBA block). When B$^+$-tree applies page compression, the 4KB-alignment constraint could incur noticeable storage space waste. This can be illustrated in Fig. 1: Assume one 16KB B$^+$-tree page is compressed to 5KB; the compressed page must occupy two LBA blocks (i.e., 8KB) on the storage device, wasting 3KB storage space. Therefore, due to the CPU overhead and storage space waste caused by the 4KB-alignment constraint, B$^+$-tree page compression is not widely used in production environment. Moreover, it is well-known that, under workloads with random writes, B$^+$-tree pages tend to be only 50%∼80% full [12]. Hence, B$^+$-tree typically has a low storage space usage efficiency. In contrast, LSM-tree has a much more compact data structure and is free from the 4KB-alignment constraint in case of compression, which leads to a higher storage space usage efficiency than B$^+$-tree.

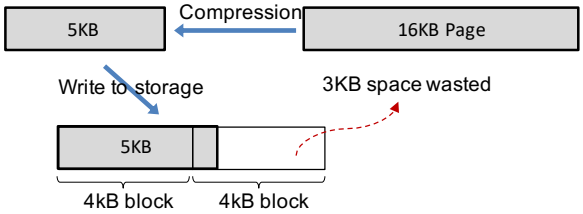

Figure 1: An example to show the storage space waste caused by 4KB-alignment constraint for B$^+$-tree page compression.

2.2 In-Storage Transparent Compression

Fig. 2 illustrates a computational storage drive (CSD) with built-in transparent compression: Inside the CSD controller chip, compression and decompression are carried out directly on the I/O path by the hardware engine, and the FTL (flash translation layer) manages the mapping of all the variable-length compressed data blocks. Since the compression is carried out inside the storage drive, it is not subject to 4KB-alignment constraint (i.e., all the compressed blocks are packed tightly in flash memory without any space waste).

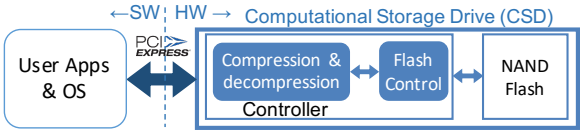

Figure 2: Illustration of a CSD with transparent compression.

As illustrated in Fig. 3, storage hardware with built-in transparent compression has the following two properties: (a) The storage hardware can expose an LBA space that is much larger than its internal physical storage capacity. This is conceptually similar to the thin provisioning. (b) Since certain data patterns (e.g., all-zero or all-one) can be highly compressed, we can leave one 4KB LBA partially filled with valid data without wasting the physical storage space. These two properties decouple the logical storage space utilization efficiency from the physical storage space utilization efficiency. This allows data management software to employ *sparse data structure* in the logical storage space without sacrificing the true physical storage cost, which creates a new spectrum of design space for data management systems [36].

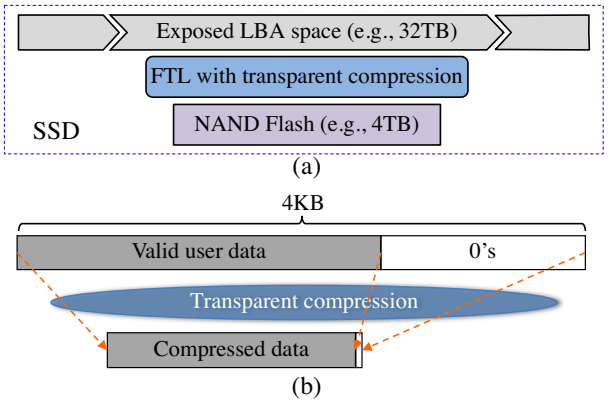

Figure 3: Illustration of the decoupled logical and physical storage space utilization efficiency enabled by storage hardware with built-in transparent compression.

2.3 B$^+$-tree vs. LSM-tree

LSM-tree has recently received significant interest (e.g., see [3, 15, 21, 22, 29, 35]) because of its advantages in terms of storage space usage and write amplification. If B+ tree has a very large cache memory (e.g., enough to hold the entire dataset) and uses very large redo log files, its write amplification could be much smaller than that of LSM-tree. Moreover, under large record size (e.g., 1KB and above), B+ tree tend to have smaller write amplification than LSM-tree. Hence, this work focuses on the scenarios where dataset is far bigger than the cache memory capacity and meanwhile the record size tends to be small (e.g., few hundred bytes or less), under which B+ tree tends to suffer from much higher write amplification than LSM-tree.

For the purpose of demonstration, we use RocksDB and WiredTiger as representatives of LSM-tree and B$^+$-tree, and carried out experiments on a 3.2TB storage drive with built-in transparent compression from ScaleFlux [31]. We run random write-only workloads with 128-byte record size over a 150GB dataset. For WiredTiger, we set its B$^+$-tree leaf page size as 8KB. Table 1 lists both the logical storage usage on the LBA space (i.e., before in-storage compression) and physical storage usage (i.e., after in-storage compression). Since LSM-tree has a more compact data structure, RocksDB has a smaller logical storage space usage than WiredTiger (i.e., 218GB vs. 280GB). Nevertheless, after in-storage transparent compression, WiredTiger consumes even less physical storage space than RocksDB, most likely due to the space amplification of LSM-tree. Fig. 4 shows the write amplification under different number of client threads. We measured the write amplification as the ratio between the volume of post-compression data being physically written to NAND flash memory inside the storage drive and the total amount of data written into database. The results show that RocksDB consistently has about 4× less write amplification than WiredTiger.

Table 1: Storage space usage comparison.

	Storage space usage	
	Logical	Physical
RocksDB	218GB	129GB
WiredTiger	280GB	104GB

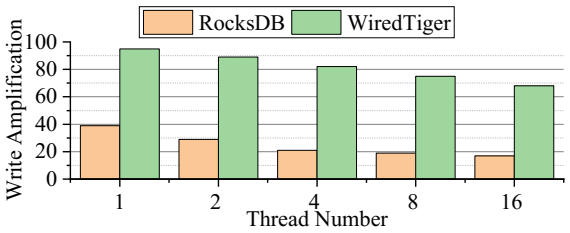

Figure 4: Measured write amplification.

The above results suggest that, with in-storage transparent compression, we could close the physical storage cost gap between B$^+$-tree and LSM-tree, while LSM-tree still maintains its significant advantage in terms of write amplification. The goal of this work is to further close the write amplification gap by appropriately modifying the B$^+$-tree implementation.

2.4 B$^+$-tree Write Amplification

Under current I/O interface protocols, storage devices only guarantee write atomicity over each 4KB LBA block. As a result, when the page size is larger than 4KB, B$^+$-tree must on its own ensure page write atomicity, which can be realized via two different strategies: (i) In-place page update: Although the convenient in-place update strategy simplifies the page storage management, B$^+$-tree must accordingly use page journaling (e.g., double-write buffer in MySQL) to survive partial page write failures, leading to about 2× higher write volume. (ii) Copy-on-write (or shadowing) page update: Although copy-on-write obviates the use of page journaling and readily supports snapshot, it complicates the page storage management. Meanwhile B$^+$-tree must employ certain mechanisms (e.g., page mapping table) to keep track of the page location, which still incurs extra storage write traffic.

Accordingly, we could classify B$^+$-tree storage write traffic into three categories: (1) logging writes that ensure transaction atomicity and isolation, (2) page writes that persist in-memory dirty B$^+$-tree pages to storage devices, and (3) extra writes that are induced by ensuring page write atomicity (e.g., page journaling in the case of in-place updates, or page mapping table persist in the case of page shadowing). Let W_{log}, W_{pg}, and W_e denote the total data write amount of these three categories, and W_{usr} denote the total amount of user data written into the B$^+$-tree. We can express the B$^+$-tree write amplification as

$$WA = \frac{W_{log}}{W_{usr}} + \frac{W_{pg}}{W_{usr}} + \frac{W_e}{W_{usr}} = WA_{log} + WA_{pg} + WA_e. \quad (1)$$

When B$^+$-tree runs on storage hardware with built-in transparent compression, let α_{log}, α_{pg}, and α_e denote the average compression ratio of the three categories of writes. Here we calculate the compression ratio by dividing the post-compression data volume with the before-compression data volume. Hence the compression ratio always falls into $(0,1]$, and a higher data compressibility leads to a smaller compression ratio. Therefore, the overall B$^+$-tree write amplification becomes

$$WA = \alpha_{log} \cdot WA_{log} + \alpha_{pg} \cdot WA_{pg} + \alpha_e \cdot WA_e. \quad (2)$$

3 Proposed Design Techniques

According to Eq. (2), we can reduce the B$^+$-tree write amplification by either reducing WA_{log}, WA_{pg}, and/or WA_e (i.e., reducing the B$^+$-tree write data volumes), or reducing α_{log}, α_{pg}, and/or α_e (i.e., improving the write data compressibility). By applying sparse data structure enabled by in-storage transparent compression, this section presents three design techniques to reduce the B$^+$-tree write amplification: (1) deterministic page shadowing that eliminates WA_e, (2) localized page modification logging that reduces both WA_{pg} and α_{pg}, and (3) sparse redo logging that reduces α_{log}.

3.1 Deterministic Page Shadowing

In order to eliminate WA_e, B$^+$-tree should employ the principle of page shadowing. Nevertheless, in conventional implementation of page shadowing, the new on-storage location of each updated B$^+$-tree page is dynamically determined during the runtime and must be recorded/persisted, leading to extra write overhead and management complexity. To eliminate the extra write overhead and meanwhile simplify the storage management, we propose a technique called *deterministic page shadowing* as illustrated in Fig. 5: Let l_{pg} denote the

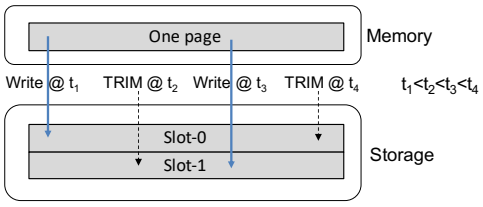

Figure 5: Illustration of deterministic page shadowing: two slots at the fixed location on the logical storage LBA space alternatively serve the memory-to-storage flush of one page.

B$^+$-tree page size (e.g., 8KB or 16KB). For each page, B$^+$-tree allocates $2l_{pg}$ amount of logical storage area on the LBA space and partitions it into two size-l_{pg} slots (slot-0 and slot-1). For each B$^+$-tree page, the two slots at the fixed location on the logical storage space serve memory-to-storage page flush alternatively in the ping-pong manner. Once a page has been flushed from memory into one slot, B$^+$-tree will issue a *TRIM* command over the other slot. This is conceptually the same as the conventional page shadowing with the difference that the location of the shadow page is now fixed. Although B$^+$-tree occupies $2\times$ larger logical storage space, only half of the storage space store valid data and the other half are trimmed (hence do not consume physical flash memory storage space). As pointed out above in Section 2.2, storage hardware with built-in transparent compression could expose a logical LBA storage space that is much larger than its internal physical storage capacity. Hence, such storage hardware can readily support the deterministic page shadowing. We note that deterministic page shadowing solely aims at ensuring page write atomicity without extra write overhead. To support multi-version concurrency control (MVCC), B$^+$-tree could use conventional methods such as undo logging.

With the proposed deterministic page shadowing, B$^+$-tree uses an in-memory bitmap to keep track of the valid slot for each page. Compared with page table being used in conventional page shadowing, bitmap consumes much less memory resource. Moreover, B$^+$-tree does not need to persist the bitmap. In case of system re-start, B$^+$-tree can gradually rebuild the in-memory bitmap: When B$^+$-tree loads one page for the first time, it reads both slots from the storage device. For the trimmed slot, storage device simply returns an all-zero block, based on which B$^+$-tree can easily identify the valid slot. When B$^+$-tree reads both slots of a page, the storage device internally only fetches the valid (i.e., untrimmed) slot from the physical storage media. Hence, compared with reading one slot, reading both slots will only incur more data transfer through the PCIe interface, without any extra read latency inside the storage device. This should not be an issue as the upcoming PCIe Gen5 will support 16GB/s$\sim$32GB/s, which is significantly larger than the back-end flash memory access bandwidth inside storage devices and hence can readily accommodate the extra data transfer. In case of system crash, B$^+$-tree needs to handle the following two possible scenarios: (i) A slot is partially written before the system crash: B$^+$-tree can easily identify the partially written slot by verifying the page checksum. (ii) A slot has been successfully written but the other slot has not been trimmed before the system crash: B$^+$-tree can identify the valid slot by comparing the page LSN (logical sequence number) of the pages on both slots. Since it is not necessary to persist the in-memory bitmap, deterministic page shadowing eliminates the $\alpha_e \cdot WA_e$ component from the total B$^+$-tree write amplification.

3.2 Localized Page Modification Logging

The second technique aims at reducing both α_{pg} and WA_{pg} components in Eq. (2). It is motivated by a simple observation: For a B$^+$-tree page, let Δ denote the difference between its in-memory image and on-storage image. If the difference is significantly smaller than the page size (i.e., $|\Delta| << l_{pg}$), we can largely reduce the write amplification by logging the

page modification Δ, instead of writing the entire in-memory page image, to the storage device. This is conceptually the same as the similarity-based data deduplication [2] and delta encoding [24]. Unfortunately, when B$^+$-tree runs on normal storage devices without built-in transparent compression, this approach is not practical due to significant operational overhead: Given the 4KB block IO interface, we must coalesce multiple Δ's from different pages into one 4KB LBA block in order to materialize the write amplification reduction. To enhance the gain, we should apply the page modification logging multiple times for each page, before resetting this process to construct the up-to-date on-storage page image. Accordingly, multiple Δ's associated with the same page will spread over multiple 4KB blocks on the storage device, which however will cause two problems: (1) For each page, B$^+$-tree must keep track of all its associated Δ's and also periodically carry out garbage collection, leading to a high storage management complexity. (2) To load a page from storage, B$^+$-tree has to read the existing on-storage page image and multiple Δ's from multiple non-contiguous 4KB LBA blocks, which leads to a long page load latency. Therefore, to our best knowledge, this simple design concept has not been used by real-world B$^+$-tree implementations ever reported in the open literature.

Storage hardware with built-in transparent compression for the first time makes the above simple idea practically viable. By applying sparse data structure enabled by such storage hardware, we no longer have to coalesce multiple Δ's from different pages into the same 4KB LBA block. Leveraging the abundant logical storage LBA space, for each B$^+$-tree page, we can simply dedicate one 4KB LBA block as its modification logging space to store the Δ, which is referred to as localized page modification logging. Under the 4KB I/O interface, to realize the proposed page modification logging for each page, B$^+$-tree writes $D = [\Delta, \mathbf{O}]$ (where $\mathbf{O}$ represents an all-zero vector, and $|D|$ is 4KB) to the 4KB block associated with the page. Inside the storage device, all the zeros in D will be compressed away and only the compressed version of Δ will be physically stored. Therefore, when serving each memory-to-storage page flush with page modification logging, we reduce WA_{pg} by writing 4KB instead of l_{pg} amount of data to the logical storage LBA space, and reduce the compression ratio α_{pg} since the written data $[\Delta, \mathbf{O}]$ can be highly compressed by the storage device. By dedicating one 4KB modification logging space for each B$^+$-tree page, we do not incur extra B$^+$-tree storage management complexity. The read amplification is small for two main reasons: (1) B$^+$-tree always reads only one additional 4KB LBA block. Moreover, each page and its associated 4KB logging block contiguously reside on the LBA space. Hence, in order to read both the page and its associated 4KB logging block, B$^+$-tree only issues a single read request to the storage device. (2) The storage device internally fetches very small amount of data from flash memory in order to reconstruct the 4KB LBA block $[\Delta, \mathbf{O}]$.

To practically implement this simple idea, B$^+$-tree must carry out two extra operations: (1) To load a page from storage into memory, B$^+$-tree must construct the up-to-date page image based on the on-storage page image and Δ. (2) To flush a page from memory to storage, B$^+$-tree must obtain Δ and accordingly decide whether it should invoke the page modification logging. To minimize the B$^+$-tree operational overhead, we propose the following implementation strategy: Let P_m and P_s denote the in-memory and on-storage images of one B$^+$-tree page. We logically partition P_m and P_s into k segments, i.e., $P_m = [P_{m,1}, \cdots, P_{m,k}]$ and $P_s = [P_{s,1}, \cdots, P_{s,k}]$, and $|P_{m,i}| = |P_{s,i}|\ \forall i$ (i.e., the two segments $P_{m,i}$ and $P_{s,i}$ at the same position have the same size). For each page, B$^+$-tree keeps a k-bit vector $f = [f_1, \cdots, f_k]$, where f_i is set to 1 if $P_{m,i} \neq P_{s,i}$. Accordingly, we construct Δ by concatenating all the in-memory segments $P_{m,i}$ with $f_i = 1$. During the runtime, whenever the i-th segment in one in-memory page is modified, B$^+$-tree will set its corresponding f_i as 1. When B$^+$-tree flushes a page from memory to storage, it first calculates the size of Δ as

$$|\Delta| = \sum_{\forall i, f_i = 1} |P_{m,i}|. \tag{3}$$

We define a fixed threshold T that is not larger than 4KB. If $|\Delta| \leq T$, then B$^+$-tree will invoke the page modification logging, where Δ can be obtained through simple memory-copy operations. We note that the k-bit vector f should be written together with Δ into the dedicated 4KB page modification logging block. When B$^+$-tree loads a page from storage into memory, it fetches $l_{pg} + 4KB$ amount of data from the storage device, where the size-l_{pg} space contains the current on-storage page image P_s and the additional 4KB block contains the associated f and Δ. Accordingly, we could easily construct the up-to-date page image through simple memory-copy operations. For each B$^+$-tree page, the size of its Δ will monotonically increase as B$^+$-tree undergoes more write operations. Once $|\Delta|$ becomes larger than the threshold T, we will reset the process by flushing the entire up-to-date page to storage with $\Delta = \emptyset$ and f being an all-zero vector. We note that the threshold T configures the trade-off between write amplification reduction and storage space amplification: As we increase the value of T, we can less frequently reset the page modification logging process, leading to a smaller write amplification. Meanwhile, under a larger value of T, more page modifications will accumulate in the logging space and cause a larger storage cost overhead.

Fig. 6 further illustrates this implementation strategy. Among the all the k segments, the first segment $P_{m,1}$ is the page header and the last segment $P_{m,k}$ is the page trailer, both of which can be much smaller than the other segments. Suppose a page update causes modification of the segment $P_{m,3}$ and page header/trailer. When B$^+$-tree evicts this page from the memory, it constructs the Δ as $[P_{m,1}, P_{m,3}, P_{m,k}]$, and writes Δ and the k-bit vector f to the dedicated 4KB block logging block, which is further compressed inside the storage device.

We note that, if B$^+$-tree treats in-memory pages as im-

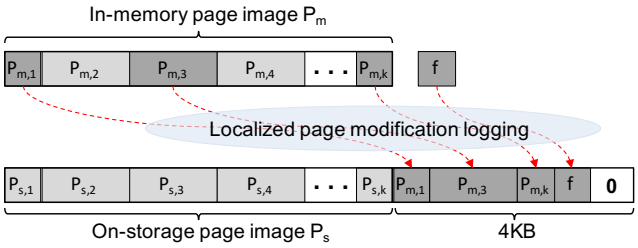

Figure 6: Illustration of the localized page modification logging, where the to-be-flushed in-memory page P_m contains three modified segments $P_{m,1}$, $P_{m,3}$, and $P_{m,k}$.

mutable and uses in-memory delta chaining to keep track of the in-memory page modification (which is used in the Bw-tree [19, 20] to achieve latch-free operations), we can most likely further reduce $|\Delta|$ and hence improve the effectiveness of the localized page modification logging on reducing the write amplification. However, such delta-chaining approach can largely complicate the B$^+$-tree implementation [32] and incur noticeable memory usage overhead. Hence, this work chooses the above simple intra-page segment-based tracking approach in our implementation and evaluation.

3.3 Sparse Redo Logging

The third design technique aims at reducing the component α_{log} in Eq. (2) (i.e., improving the redo log data compressibility). To maximize the reliability, B$^+$-tree flushes the redo log with *fsync* or *fdatasync* at every transaction commit. In order to reduce the log-induced storage overhead, conventional practice always tightly packs log records into the redo log. As a result, multiple consecutive redo log flushes may write to the same LBA block on the storage device, especially when transaction records are significantly smaller than 4KB and/or the workload concurrency is not very high. This can be illustrated in Fig. 7: Suppose three transactions TRX-1, TRX-2, and TRX-3 (with log records L_1, L_2, and L_3) commit at the time t_1, t_2, and t_3, respectively, where $t_1 < t_2 < t_3$. As illustrated in Fig. 7, at the time t_1, 4KB data $[L_1, \mathbf{O}]$ is flushed from the in-memory redo log buffer to the LBA 0x0001 on the storage device that further internally compresses the data. Later on, the log record L_2 is appended into the redo log buffer, and at the time t_2, the 4KB data $[L_1, L_2, \mathbf{O}]$ is flushed to the same LBA 0x0001 on the storage device. Similarly, at the time t_3, the 4KB data $[L_1, L_2, L_3, \mathbf{O}]$ is flushed to the same LBA 0x0001 on the storage device. As illustrated in Fig. 7, the same log record (e.g., L_1 and L_2) are written to the storage device multiple times, leading to a higher write amplification. Equivalently, as more log records are accumulated inside each 4KB redo log buffer block, the redo log data compression ratio α_{log} will become worse and worse over the multiple consecutive redo log flushes.

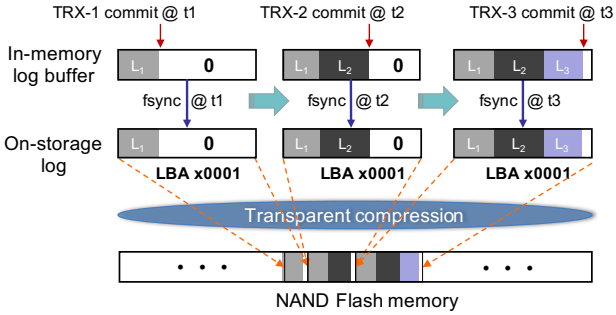

Figure 7: Conventional implementation of redo logging where log records are tightly packed into redo log and consecutive transactions commits could flush redo log to the same LBA (e.g., LBA 0x0001 in this example) multiple times.

By applying sparse data structure enabled by storage hardware with built-in transparent compression, we propose a design technique called sparse redo logging that can enable the storage hardware most effectively compress the redo log and hence reduce the logging-induced write amplification. Its basic idea is very simple: At each transaction commit and its corresponding redo log memory-to-storage flush, we always pad zeros into the in-memory redo log buffer to make its content 4KB-aligned. As a result, the next log record will be written into a new 4KB space in the redo log buffer. Therefore, each log record will be written to the storage device only once, leading to a lower write amplification compared with the conventional practice. This can be further illustrated in Fig. 8: Assuming the same scenario as shown above in Fig. 7, after the transaction TRX-1 commits at the time t_1, we pad zeros into the redo log buffer and flush the 4KB data $[L_1, \mathbf{O}]$ to the LBA 0x0001 on the storage device. Subsequently, we put the next log record L_2 in a new 4KB space in the redo log buffer. At the time t_2, the 4KB data $[L_2, \mathbf{O}]$ is flushed to a new LBA 0x0002 on the storage device. Similarly, at the time t_3, the 4KB data $[L_3, \mathbf{O}]$ is flushed to another new LBA 0x0003 on the storage device. Clearly, each redo log record

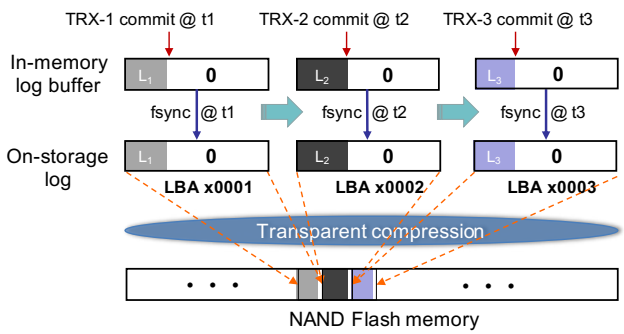

Figure 8: Illustration of the proposed sparse logging where each redo log flush always writes to a new LBA block.

is written to the storage device only once, and redo log writes can be (much) better compressed by the storage hardware, leading to a (much) smaller α_{log} and hence lower write amplification. Since each transaction commit always invokes one 4KB write to the storage device in both conventional logging and proposed sparse logging, the total redo log write volume W_{log} in Eq. (2) will remain the same. Therefore, by reducing the log compression ratio α_{log}, the proposed sparse logging reduces the component $\alpha_{log} \cdot W_{log}$ in the total B$^+$-tree write amplification.

4 Evaluation

For the purpose of demonstration, we implemented a B$^+$-tree (referred to as B$^-$-tree) that incorporates our proposed three simple design techniques. To facilitate the comparison, we also implemented a baseline B$^+$-tree that uses the conventional page shadowing, where we persist the page table after each page flush. Since the proposed three design techniques mainly confine within the I/O module and are largely orthogonal to the other modules in B$^+$-tree implementation, we obtained the B$^-$-tree by simply integrating the proposed design techniques into the baseline B$^+$-tree with 1,200 LoC added/modified. Moreover, we also considered RocksDB and WiredTiger as representatives of LSM-tree and normal B$^+$-tree. For RocksDB, we set its maximum number of compaction and flush threads as 12 and 4, and set the Bloomfilter as 10 bits per record. For WiredTiger and our own baseline B$^+$-tree and B$^-$-tree, we use 4 background write threads that flush dirty in-memory pages to the storage device.

4.1 Experimental Setup

We ran all the experiments on a server with 24-core 2.6GHz Intel CPU, 64GB DDR4 DRAM, and a 3.2TB ScaleFlux computational storage drive with built-in transparent compression. This 3.2TB drive carries out hardware-based zlib compression on each 4KB block directly along the internal I/O path, being transparent to the host. The per-4KB (de)compression latency of the hardware zlib engine is around 5μs, which is over 10$\times$ shorter than the TLC/QLC NAND flash memory read latency ($\sim$50μs and above) and write latency ($\sim$1ms and above). Operating with PCIe Gen3$\times$4 interface, this computational storage drive can achieve up to 3.2GB/s sequential throughput and 650K (520K) random 4KB read (write) IOPS (I/O per second) over 100% LBA span. In comparison, leading-edge commodity NVMe SSDs (e.g., Intel P4610) achieve similar sequential throughput and random 4KB read IOPS, but have much worse random 4KB write IOPS (e.g., below 300K). This is because built-in transparent compression can significantly reduce the garbage collection overhead inside the storage drive. This computational storage drive is already in volume production and has been deployed in data centers worldwide.

This computational storage drive can report the amount of post-compression data being physically written to the NAND flash memory, which are used in the calculation of write amplification. Before measuring the write amplification for each case, we populate the B$^+$-tree/LSM-tree data store by inserting all the data records in a fully random order. Once after the data store has been fully populated, we subsequently run random write-only workloads over one hour in order to measure the write amplification. In all our experiments, we generate the content of each record by filling its half content as all-zero and the other half content as random bytes in order to mimic the runtime data content compressibility.

We note that the effectiveness of the proposed sparse redo logging strongly depends on the redo log flush policy. As discussed above Section 3.3, when redo log flushes at every transaction commit to maximize the system reliability, sparse redo logging is very effective. However, for applications that can tolerate the loss of certain amount of most recent data, one could relax the redo log flush policy (e.g., flush every one minute) under which the proposed sparse redo logging will be much less useful. Therefore, we considered two scenarios in our evaluation: (1) redo log flush per transaction commit (denoted as *log-flush-per-commit*), and (2) redo log flush per minute (denoted as *log-flush-per-minute*).

4.2 Experiments with Log-Flush-Per-Minute

We first carried out experiments without taking into account of the benefit of sparse redo logging by setting the redo log flush policy as per-minute. We considered two different dataset size: (1) 150GB dataset with 1GB cache memory, and (2) 500GB dataset with 15GB cache memory. We also considered three different record size (including 8B key): 128B, 32B, and 16B. For B$^+$-tree implementations, following the popular RDBMs such as Oracle and MySQL, we considered two different page size, including 8KB and 16KB. For our B$^-$-tree, the implementation of the proposed page modification logging involves the following two parameters: (1) the threshold T that determines the maximum $|\Delta|$ per page, and (2) the segment size (denoted as D_s) when partitioning each page into multiple segments for tracking page modification, as discussed in Section 3.2.

Fig. 9 and Fig. 10 show the measured write amplification for 150GB and 500GB datasets, respectively. In each experiment, we use either 1, 2, 4, 8, or 16 client threads to cover a wide range of runtime workload concurrency. For B$^-$-tree, we set the threshold T as 2KB, and set the segment size D_s as either 128B or 256B. Since both WiredTiger and our own baseline B$^+$-tree use page shadowing, they have very similar write amplification as shown in Fig. 9 and Fig. 10. Compared with RocksDB, normal B$^+$-tree (i.e., WiredTiger and our own baseline B$^+$-tree) has a much larger write amplification, while our B$^-$-tree can essentially close the B$^+$-tree vs. LSM-tree write amplification gap. For example, in the case of 500GB dataset and 32B record size and 4 client threads, the write am-

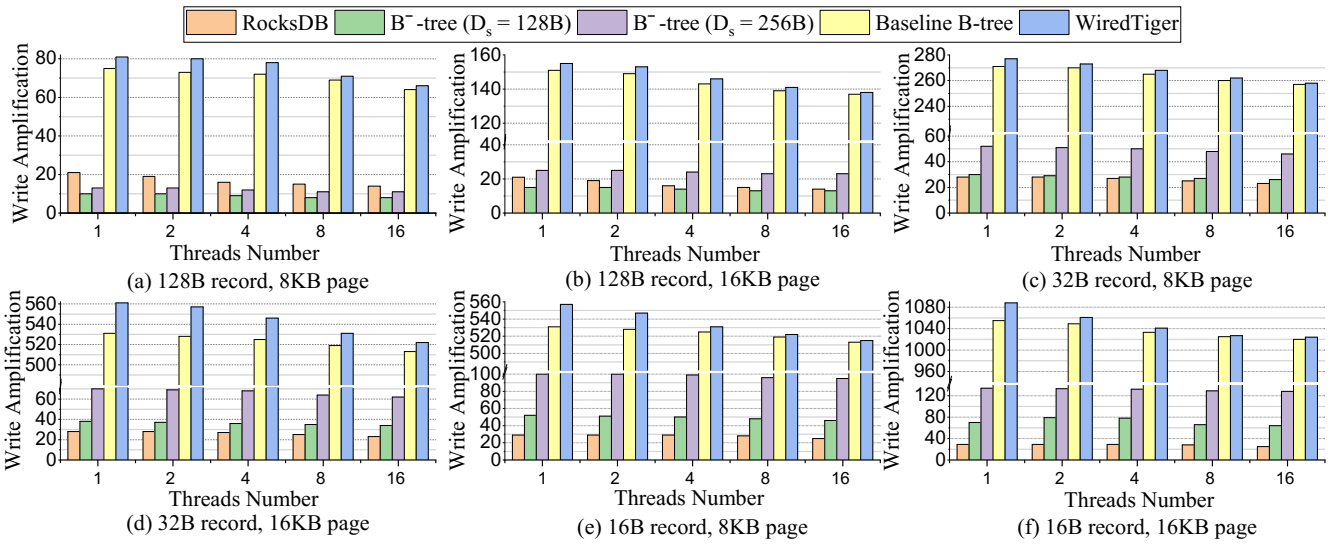

Figure 9: Write amplification under the log-flush-per-minute policy, where the dataset size is 150GB and cache size is 1GB.

plification of RocksDB is 38, while the write amplification of WiredTiger is 268 under 8KB page size and 530 under 16KB page size, respectively, which are 7.1× and 13.9× larger than that of RocksDB. In comparison, the write amplification of B$^-$-tree with D_s=128B is 28 under 8KB page size (which is only 73.7% of RocksDB's write amplification) and 36 under 16KB page size (which is almost the same as RocksDB).

As shown in both Fig. 9 and Fig. 10, the write amplification of both normal B$^+$-tree and B$^-$-tree will increase as we reduce the record size (e.g., from 128B per record to 16B per record) and/or increase the B$^+$-tree page size (i.e., from 8KB to 16KB). Since we use the log-flush-per-minute policy, the overall write amplification of both normal B$^+$-tree and B$^-$-tree tends to be dominated by the $\alpha_{pg} \cdot WA_{pg}$, as shown in Eq. (2). In the case of normal B$^+$-tree, WA_{pg} proportionally increases as we reduce the record size and/or increase the page size. Therefore, the write amplification of normal B$^+$-tree almost linearly scale with the page size and the inverse of the record size. In the case of B$^-$-tree, its $\alpha_{pg} \cdot WA_{pg}$ not only depends on the record size and page size, but also depends on the threshold T and segment size D_s. Hence, the write amplification of B$^-$-tree tends to sub-linearly scale with the page size and the inverse of the record size, as shown in both Fig. 9 and Fig. 10. In contrast, due to the nature of LSM-tree, the write amplification of RocksDB is weakly dependent on the record size.

As the number of client threads increases, the write amplification of normal B$^+$-tree noticeably reduces, because of the larger probability of page flush coalescing under higher workload concurrency. In comparison, the write amplification of B$^-$-tree is much more weakly dependent on the number of client threads, because the probability that different client threads modify the same segment inside a page is much smaller than the probability that different client threads modify the same page. Moreover, the write amplification of B$^-$-tree increases as we increase the segment size D_s, simply because the page modification logging is done in the unit of segments. The impact of segment size D_s on the write amplification is more significant under smaller record size, as shown in both Fig. 9 and Fig. 10.

The write amplification of LSM-tree may noticeably increase as the dataset size increases, which can be observed by comparing the results in Fig. 9 and Fig. 10. This is because a larger dataset size results in more levels in LSM-tree, while the write amplification of LSM-tree tends to be proportional to the number of levels. In contrast, the write amplification of B$^+$-tree is very weakly dependent on the dataset size. As a result, the write amplification comparison of RocksDB vs. B$^-$-tree is noticeably different between the 150GB dataset and 500GB dataset. In the case of 150GB dataset as shown in Fig. 9, the write amplification of RocksDB can be up to 2× larger than that of B$^-$-tree (under 128B per record and 8KB page size), and can be up to 4× smaller than that of B$^-$-tree (under 16B per record and 16KB page size). In comparison, in the case of 500GB dataset as shown in Fig. 10, the write amplification of RocksDB can be up to 3× larger than that of B$^-$-tree (under 128B per record and 8KB page size), and can be up to 2× smaller than that of B$^-$-tree (under 16B per record and 16KB page size). The results clearly show that, even without taking into account of the effectiveness of sparse redo logging, the proposed B$^-$-tree can already close the write amplification gap between B$^+$-tree and LSM-tree.

4.3 Experiments with Log-Flush-Per-Commit

We carried out further experiments by switching to the log-flush-per-commit policy, under which the proposed sparse

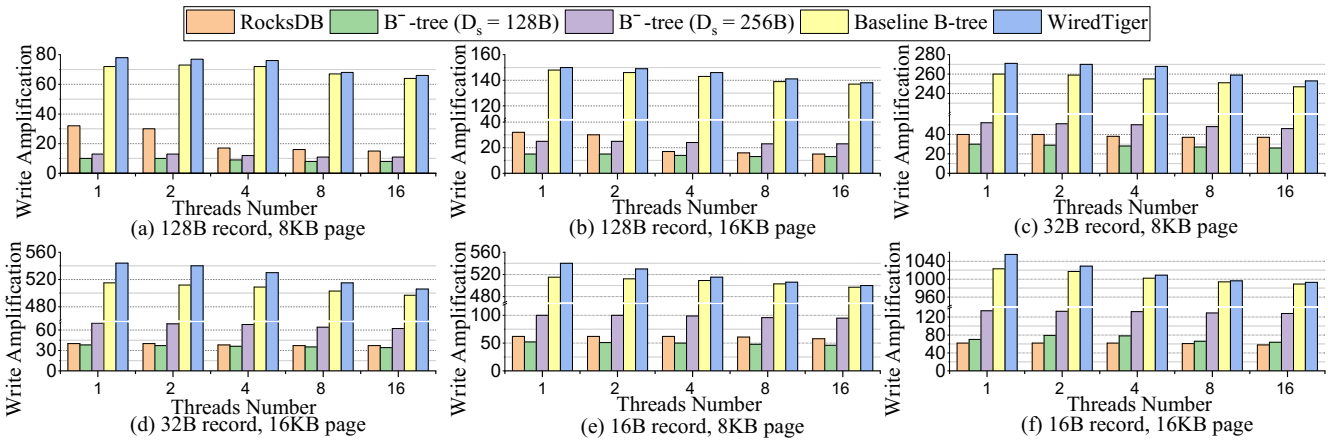

Figure 10: Write amplification under the log-flush-per-minute policy, where the dataset size is 500GB and cache size is 15GB.

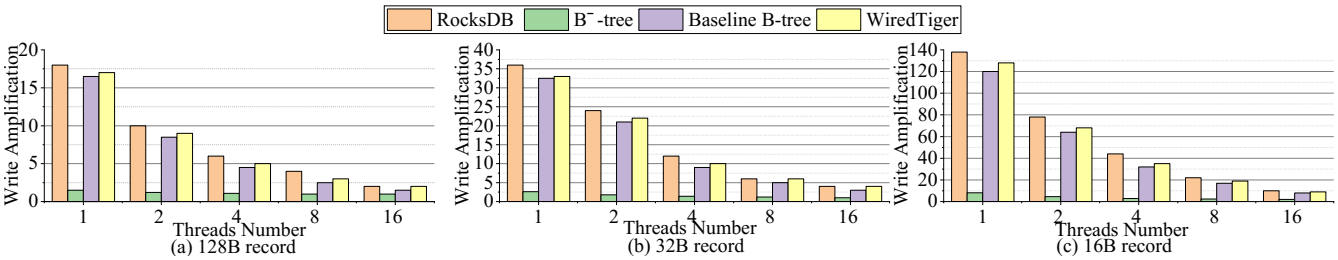

Figure 11: Log-induced write amplification when using the log-flush-per-commit policy.

redo logging can noticeably contribute to reducing the write amplification. First, Fig. 11 shows the measured write amplification caused by the log flush, i.e., the $\alpha_{log} \cdot WA_{log}$ component in Eq. 2. Given the record size, except the case of B$^-$-tree, the log-induced write amplification significantly reduces as we increase the number of client threads. This is because, under higher workload concurrency, more transaction commits can be coalesced in each log flush. In contrast, the log-induced write amplification of B$^-$-tree is much more weakly dependent on the number of client threads, because of its use of the sparse redo logging. As the record size reduces, the log-induced write amplification almost proportionally increases when the sparse redo logging is not being used. The results in Fig. 11 clearly demonstrate the effectiveness of the proposed sparse redo logging design technique when data management systems use the log-flush-per-commit policy to improve the data reliability.

Fig. 12 further shows the total write amplification under the log-flush-per-commit policy, where the dataset size is 150GB and cache size is 1GB. Compared with the experiments under the log-flush-per-minute policy (as shown in Fig. 9), the write amplification of B$^-$-tree remains almost the same, while the write amplification of the other three cases (i.e., RocksDB, our own baseline B$^+$-tree, and WiredTiger) noticeably increases, especially when the number of client threads is small, because of the higher log-induced write amplification. As a result, B$^-$-tree can more effectively close the B$^+$-tree vs. LSM-tree write amplification gap and be able to achieve better-than-RocksDB write amplification under more scenarios.

4.4 Impact of Threshold T

As discussed earlier in Section 3.2, the proposed page modification logging design approach is subject to a write amplification vs. storage usage trade-off that is configured by the threshold $T \in (0, 4\text{KB}]$. As we increase the value of T, we can pack more modification logs into each dedicated 4KB log space in order to further reduce the total write amplification, which nevertheless meanwhile induces higher storage usage overhead. All the experiments above were carried out with T as 2KB. We carried out further experiments under different values of threshold T to study its impact on the write amplification vs. storage usage trade-off. For each B$^+$-tree page P_i, let $|\Delta_i|$ denote the size of its associated modification log. Let N denote the total number of B$^+$-tree pages and recall that l_{pg} denotes the page size, we can express the average storage usage overhead factor as

$$\beta = \frac{\sum_{i=1}^{N} |\Delta_i|}{N \cdot l_{pg}}. \quad (4)$$

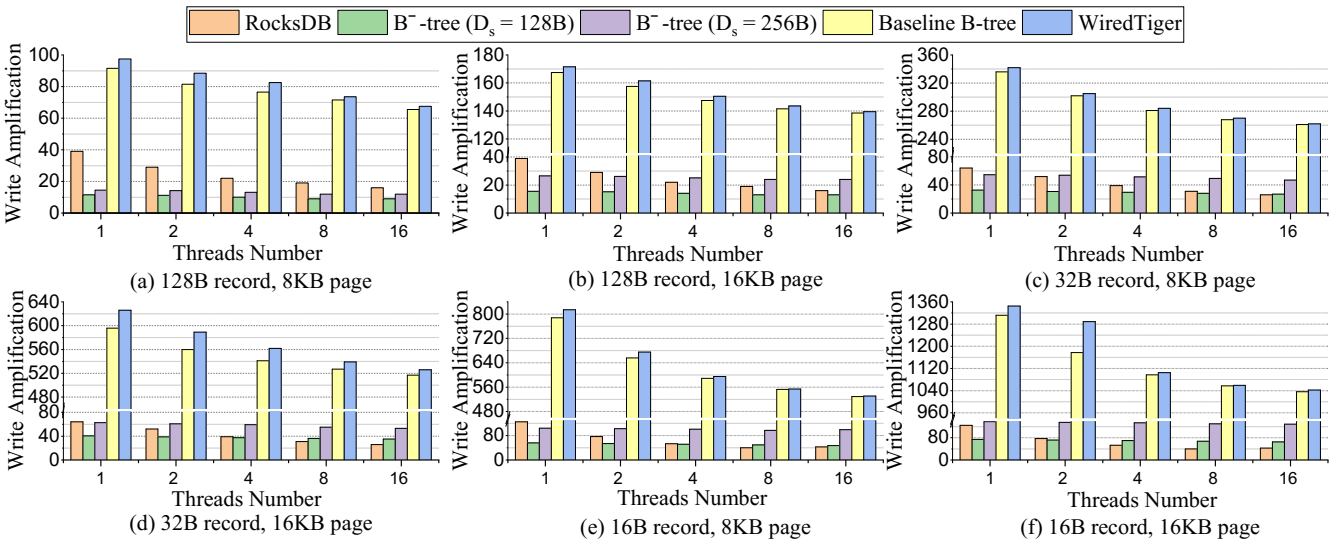

Figure 12: Write amplification under the log-flush-per-commit policy, where the dataset size is 150GB and cache size is 1GB.

Under a sufficiently large N, the value of β mainly depends on the page size l_{pg}, the threshold T, and the workload characteristics (in particular the write request distribution over all the pages). It also weakly depends on the segment size D_s. Assuming the fully random write request distribution across all the pages, we carried out experiments to measure the average value of β, and the results are summarized below in Table 2. The results clearly show that the storage usage overhead will reduce as we reduce the threshold T and/or increase the page size. In comparison, the impact of the segment size D_s is much more insignificant.

Table 2: Storage usage overhead factor β of B⁻-tree.

Page size	D_s	Threshold T		
		4KB	2KB	1KB
8KB	128B	27.0%	12.4%	5.6%
	256B	26.3%	11.5%	4.8%
16KB	128B	12.7%	6.0%	2.8%
	256B	12.3%	5.6%	2.3%

Fig. 13 further compares the total storage usage in terms of both logical storage usage on the LBA space (i.e., before in-storage compression) and physical usage of flash memory (i.e., after in-storage compression). Since LSM-tree has a more compact data structure than B⁺-tree, RocksDB has a (much) smaller logical storage usage than the others as shown in Fig. 13. Since B⁻-tree allocates one 4KB block for each page in order to implement the localized modification logging, its logical storage usage is much larger than that of normal B⁺-tree. Nevertheless, after the in-storage compression, WiredTiger and our baseline B⁺-tree consume less physical flash memory capacity than RocksDB (most likely because of the space amplification of LSM-tree) and B⁻-tree (because of the storage overhead caused by page modification logging). Due to the storage space overhead caused by page modification logging, B⁻-tree has slightly larger physical storage usage than RocksDB. For example, in the case of 500GB dataset size, the physical storage usage of RocksDB is 431GB, while the physical storage usage of B⁻-tree with T=2KB is 452GB, only about 5% larger than that of RocksDB.

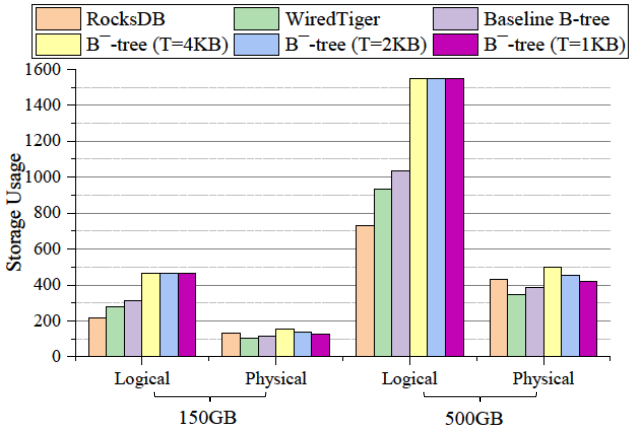

Figure 13: Comparison of logical and physical storage space usage where B⁺-tree page size is 8KB.

Fig. 14 compares the write amplification of B⁻-tree under different value of the threshold T, where we use the log-flush-per-minute policy in order to better show the impact of T. The segment size D_s is 128B. The results clearly show that we can reduce the write amplification by increasing the threshold T. Moreover, the reduction on the write amplification tends to become less and less as we continue to increase the threshold T. This is because, as the page modification log size $|\Delta|$

becomes larger, the write amplification caused by flushing the modification log will accordingly increase. Combining the results shown in Fig. 13 and Fig. 14, we can observe the impact of the threshold T on the trade-off between the write amplification and storage usage overhead. The setting of T=2KB appears to achieve a reasonable balance on the trade-off and hence has been used in all the experiments presented above in Sections 4.2 and 4.3.

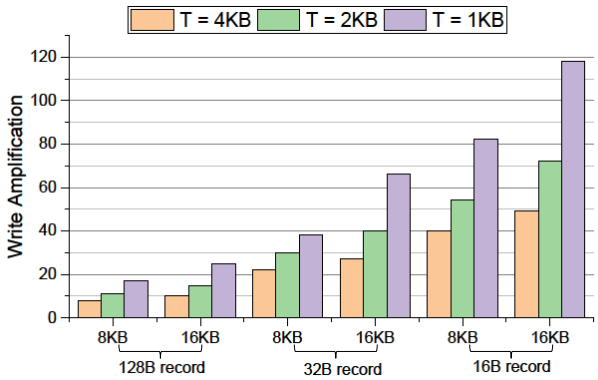

Figure 14: B$^-$-tree write amplification under different T.

4.5 Speed Performance Evaluation

Finally, we studied the speed performance of B$^-$-tree. Compared with normal B$^+$-tree, B$^-$-tree tends to have lower read speed performance because of the following two overheads when fetching each page from the storage: (1) B$^-$-tree has to fetch an extra 4KB block from the storage, and (2) B$^-$-tree has to consolidate the modification log with the current on-storage page image in order to construct the up-to-date in-memory page image. Using the 150GB dataset with 128B per record as the test vehicle, we run random read-only workloads with either point read or range scan queries. The B$^+$-tree page size is 8KB in all the experiments. Fig. 15 shows the measured TPS performance under random point read queries. The results show that normal B$^+$-tree (WiredTiger and our own baseline B$^+$-tree) have the best point read throughput performance. RocksDB and B$^-$-tree achieve almost the same random point read throughput performance. By using the Bloomfilter, RocksDB almost completely obviates the read amplification problem of classical LSM-tree. Nevertheless, when serving read requests, RocksDB still has to search the memtable and check the Bloomfilter. As shown in Fig. 15, the point read throughput gap between normal B$^+$-tree and RocksDB/B$^-$-tree is not significant. For example, under 16 client threads, WiredTiger can achieve 71K TPS, while RocksDB/B$^-$-tree can achieve 57K TPS, about 19.7% less than that of WiredTiger.

Fig. 16 shows the measured TPS when running random range scan queries, where each range scan covers 100 consecutive records. Compared with the case of random point reads, the normal B$^+$-tree and B$^-$-tree have noticeably smaller difference in terms of range scan throughput performance. This is because the two overheads of B$^-$-tree (i.e., fetching an extra 4KB, and in-memory page reconstruction) can be amortized among the records covered by each range scan. In comparison, RocksDB has noticeably worse range scan throughput performance than the others, because range scan invokes reads over all the levels in LSM-tree, leading to very high read amplification.

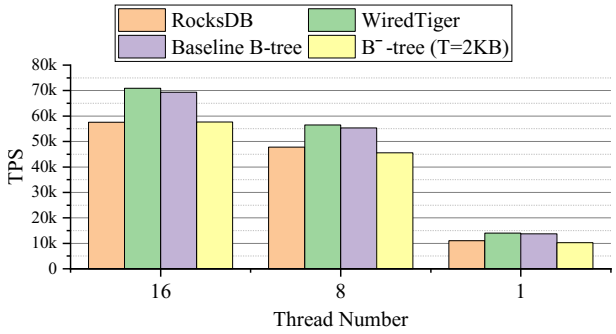

Figure 15: Random point read speed performance measured on 150GB dataset with 1GB cache and 128B per record.

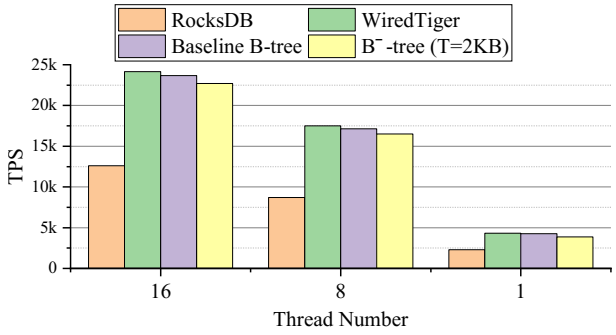

Figure 16: Random range scan speed performance measured on 150GB dataset with 1GB cache and 128B per record, where each range scan covers 100 consecutive records.

We also studied the speed performance under random write-only workloads. The random write speed performance of B$^+$-tree and LSM-tree is fundamentally limited by the write amplification. Therefore, by significantly reducing the write amplification, B$^-$-tree should be able to achieve much higher write speed performance. Fig. 17 shows the measured random write TPS on 150GB dataset with 128B per record, where the B$^+$-tree page size is 8KB. We set the log-flush-per-minute policy in the experiments. Even without the help of the sparse redo logging, B$^-$-tree achieves 19% higher write throughput than RocksDB, and about 2.1× higher write throughput than WiredTiger and our baseline B$^+$-tree. Although the workload

is write-only, the I/O traffic is heavily read/write-mixed because the cache memory capacity is much smaller than the total dataset. Because the localized page modification logging invokes read-modify-write operations, our B$^-$-tree incurs higher read I/O traffic than normal B$^+$-tree. As a result, the TPS gain of our B$^+$-tree is less than the WA reduction of B$^+$-tree as shown above in Fig. 9. Nevertheless, the random write speed results still correlate with the write amplification results, and our B$^+$-tree can achieve the highest random write speed performance.

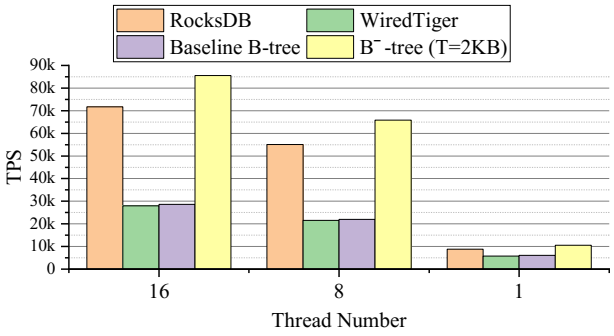

Figure 17: Random write speed performance measured on 150GB dataset with 1GB cache and 128B per record.

5 Related Work

Graefe [11] surveyed a variety of design techniques (e.g., I/O optimization, buffering, and relaxing transaction guarantee) that can improve the B$^+$-tree write throughput, some of which accomplish the goal by reducing the B$^+$-tree write amplification. Nevertheless, I/O optimization techniques that mainly aim at converting random page writes to sequential page writes are only useful to HDDs, since modern SSDs achieve almost the same random vs. sequential write speed performance. Many techniques surveyed in [11] (e.g., buffering, relaxing transaction guarantee) are orthogonal to the solutions presented in this paper, and hence can be applied altogether to further reduce the B$^+$-tree write amplification. Moreover, copy-on-write or page shadowing [1, 18] is a well-known technique to achieve B$^+$-tree data atomicity and durability. Compared with B$^+$-tree using in-place update, it can reduce the write amplification by about 2×.

Levandoski *et al.* [19, 20] proposed the Bw-tree that can better adapt to modern multi-core CPU architecture and meanwhile reduce the write amplification. Bw-tree treats each in-memory page as immutable and uses delta chaining to keep track of the changes made to each page. This can enable latch-free operations and hence better utilize multi-core CPUs. Meanwhile, by only flushing the delta records, Bw-tree can reduce the write amplification. Bw-tree uses a log-structured store to persist all the pages and deltas, which however suffers from read amplification and background garbage collection overheads. When running Bw-tree on storage hardware with build-in transparent compression, one could enhance Bw-tree by replacing the log-structured store with the localized page modification logging presented in this work.

B$^\varepsilon$-tree [4] is another well-known variant of B$^+$-tree that can significantly reduce the write amplification through data buffering at non-leaf nodes. It has been used in the design of filesystem [10, 16, 17, 34] and key-value store [8, 26]. In essence, B$^\varepsilon$-tree cleverly mixes the key design principles of B$^+$-tree and LSM-tree. Similar to LSM-tree, B$^\varepsilon$-tree has worse range scan speed performance than B$^+$-tree. Percona TokuDB [27] is one publicly known database product that is built upon B$^\varepsilon$-tree.

Little prior research has been done on studying how data management systems could take advantage of modern storage hardware with built-in transparent compression. Recently, Zheng *et al.* [36] discussed some possible options on leveraging such modern storage hardware to improve data management software design. Chen *et al.* [6] presented a hash-based key-value store that can leverage such modern storage hardware to obviate the use of costly in-memory hash table.

6 Conclusions

This paper presents three simple yet effective design techniques that enable B$^+$-tree take better advantages of modern storage hardware with built-in transparent compression. By decoupling logical vs. physical storage space utilization efficiency, such modern storage hardware allows data management systems employ sparse data structure without sacrificing the true physical data storage cost. This opens a new but largely unexplored spectrum of opportunities to innovate data management software design. As one small step towards exploring this design spectrum, this paper presents three design techniques that can appropriately embed sparsity into B$^+$-tree data structure to largely reduce the B$^+$-tree write amplification. Experimental results show that the proposed design techniques can reduce the B$^+$-tree write amplification by over 10×, which essentially closes the B$^+$-tree vs. LSM-tree gap in terms of write amplification. This work suggests that the arrival of such new storage hardware warrants a revisit on the role and comparison of B$^+$-tree and LSM-tree in future data management systems.

Acknowledgments

We would like to thank our shepherd Randal Burns and the anonymous reviewers for their insight and suggestions that help us to improve the quality and presentation of this paper. This work was supported by the National Science Foundation under Grant No. CNS-2006617.

References

[1] R. Agrawal and D. J. Dewitt. Integrated concurrency control and recovery mechanisms: Design and performance evaluation. *ACM Transactions on Database Systems (TODS)*, 10(4):529–564, 1985.

[2] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of ACM International Systems and Storage Conference (SYSTOR)*, pages 1–14, 2009.

[3] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 363–375, 2017.

[4] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, volume 3, pages 546–554. Citeseer, 2003.

[5] I. Burstein. Nvidia Data Center Processing Unit (DPU) Architecture. In *IEEE Hot Chips Symposium (HCS)*, pages 1–20, 2021.

[6] X. Chen, N. Zheng, S. Xu, Y. Qiao, Y. Liu, J. Li, and T. Zhang. KallaxDB: A table-less hash-based key-value store on storage hardware with built-in transparent compression. In *Proceedings of the International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–10, 2021.

[7] D. Chiou, E. Chung, and S. Carrie. (Cloud) Acceleration at Microsoft. *Tutorial at Hot Chips*, 2019.

[8] A. Conway, A. Gupta, V. Chidambaram, M. Farach-Colton, R. Spillane, A. Tai, and R. Johnson. SplinterDB: Closing the bandwidth gap for nvme key-value stores. In *USENIX Annual Technical Conference (ATC)*, pages 49–63, 2020.

[9] Dell EMC PowerMax. https://delltechnologies.com/.

[10] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *HotStorage*, 2012.

[11] G. Graefe. B-tree indexes for high update rates. *ACM Sigmod Record*, 35(1):39–44, 2006.

[12] G. Graefe and H. Kuno. Modern B-tree techniques. In *IEEE International Conference on Data Engineering*, pages 1370–1373. IEEE, 2011.

[13] E. F. Haratsch. SSD with Compression: Implementation, Interface and Use Case. In *Flash Memory Summit*, 2019.

[14] HPE Nimble Storage. https://www.hpe.com/.

[15] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 651–665. ACM, 2019.

[16] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: A right-optimized write-optimized file system. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 301–315, 2015.

[17] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: Write-optimization in a kernel file system. *ACM Transactions on Storage (TOS)*, 11(4):1–29, 2015.

[18] J. Kent, H. Garcia-Molina, and J. Chung. An experimental evaluation of crash recovery machanisms. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 113–122, 1985.

[19] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *IEEE International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

[20] J. J. Levandoski, S. Sengupta, and W. Redmond. The Bw-tree: A latch-free B-tree for log-structured flash storage. *IEEE Data Eng. Bull.*, 36(2):56–62, 2013.

[21] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):5, 2017.

[22] C. Luo and M. Carey. LSM-based storage techniques: a survey. *The VLDB Journal*, 29:393–418, 2020.

[23] LZ4. https://github.com/lz4/.

[24] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM conference on Applications, technologies, architectures, and protocols for computer communication*, pages 181–194, 1997.

[25] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[26] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and implementation of a fast

and efficient scale-up key-value store. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 537–550, 2016.

[27] Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[28] Pure Storage FlashBlade. https://purestorage.com/.

[29] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 497–514, 2017.

[30] RocksDB. https://github.com/facebook/rocksdb.

[31] ScaleFlux Computational Storage. http://scaleflux.com.

[32] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 473–488, 2018.

[33] WiredTiger. https://github.com/wiredtiger/.

[34] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, and M. Bender. Optimizing every operation in a write-optimized file system. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2016.

[35] Y. Yue, B. He, Y. Li, and W. Wang. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):961–973, 2016.

[36] N. Zheng, X. Chen, J. Li, Q. Wu, Y. Liu, Y. Peng, F. Sun, H. Zhong, and T. Zhang. Re-think data management software design upon the arrival of storage hardware with built-in transparent compression. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2020.

[37] zlib. http://zlib.net.

[38] Zstandard (ZSTD). https://github.com/facebook/zstd.