Breathing New Life into An Old Tree: Resolving Logging Dilemma of B^+-tree on Modern Computational Storage Drives

Kecheng Huang
The Chinese University of Hong Kong
kchuang21@cse.cuhk.edu.hk

Zhaoyan Shen*
Shandong University
shenzhaoyan@sdu.edu.cn

Zili Shao
The Chinese University of Hong Kong
shao@cse.cuhk.edu.hk

Tong Zhang
Rensselaer Polytechnic Institute
ScaleFlux Inc.
tzhang@ecse.rpi.edu

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

ABSTRACT

Having dominated databases and various data management systems for decades, B^+-tree is infamously subject to a logging dilemma: One could improve B^+-tree speed performance by equipping it with a larger log, which nevertheless will degrade its crash recovery speed. Such a logging dilemma is particularly prominent in the presence of modern workloads that involve intensive small writes. In this paper, we propose a novel solution, called *per-page logging* based B^+-tree, which leverages the emerging computational storage drive (CSD) with built-in transparent compression to fundamentally resolve the logging dilemma. Our key idea is to divide the large single log into many small (e.g., 4KB), highly compressible *per-page logs*, each being statically bounded with a B^+-tree page. All per-page logs together form a very large over-provisioned log space for B^+-tree to improve its operational speed performance. Meanwhile, during crash recovery, B^+-tree does not need to scan any per-page logs, leading to a recovery latency independent from the total log size. We have developed and open-sourced a fully functional prototype. Our evaluation results show that, under small-write intensive workloads, our design solution can improve B^+-tree operational throughput by up to 625.6% and maintain a crash recovery time of as low as 19.2 ms, while incurring a minimal storage overhead of only 0.5-1.6%.

PVLDB Reference Format:
Kecheng Huang, Zhaoyan Shen, Zili Shao, Tong Zhang, and Feng Chen. Breathing New Life into An Old Tree: Resolving Logging Dilemma of B^+-tree on Modern Computational Storage Drives. PVLDB, 17(2): 134 - 147, 2023.
doi:10.14778/3626292.3626297

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

1 INTRODUCTION

Since the 1970s, the B^+-tree index has gained widespread adoption and becomes the dominant data indexing structure in a wide range of data-centric applications, serving as a cornerstone in databases, search engines, file systems, web caches, and many other systems [6, 8, 11, 27, 34, 36, 42, 45, 49, 54, 57, 59, 62, 65, 68, 69, 75, 79]. B^+-tree has many important and highly desirable merits, such as the efficient sequential traversal operations by keeping all keys in the sorted order, a hierarchical structure that minimizes the number of random reads during lookup operations, partially filled data blocks that can speed up insertions and deletions, and a recursive algorithm that ensures the index structure remains balanced even with frequent changes. These features make B^+-tree a highly efficient, scalable, and performant structure for indexing large datasets and enabling fast queries [17, 32, 46, 61, 74, 85].

However, in recent years, the emergence of modern Internet-based services and web applications has presented unprecedented challenges for the classic B^+-tree structure. One of the most critical challenges is its frequently-criticized inability to handle workloads that involve a large amount of small-sized writes (less than 100 bytes) [3, 19, 43, 46, 85], which is unfortunately often the case in today's Internet workloads. As a result, many application systems are abandoning the once-successful B^+-tree structure, despite its many well-established and recognized merits.

1.1 Critical Issues

The limitations of B^+-tree are deeply rooted in its basic structural design. As a multi-tier balanced tree structure, B^+-tree stores data in leaf nodes and keeps data strictly sorted, which enables quick locating of the target data. However, such an indexing structure has an inherent limitation—the data records must be updated in place, which leads to a sequence of critical issues that make it difficult for B^+-tree to handle workloads with intensive small writes.

Issue #1: I/O amplification. In a typical B^+-tree-based system, data records are stored in *pages*, which are often much larger than a data record. When a record (e.g., 100 bytes) is updated, the entire data page (e.g., 16 KB) must be first loaded from storage, updated in memory, and then written back to persist the change on storage. This process, known as "read-modify-write", can cause severe *I/O amplification* problems [46, 68, 81, 85], resulting in a storm of random I/Os with unnecessary data transfer. Moreover, since most B^+-trees tend to retain a low fill factor (the ratio of valid data to node size) to reduce the number of index node splits, most of these data pages contain only half a page of meaningful data, which is a further waste of both I/O time and memory.

*Corresponding author

Proceedings of the VLDB Endowment, Vol. 17, No. 2 ISSN 2150-8097.
doi:10.14778/3626292.3626297

Issue #2: Underutilized buffer pool. In an attempt to address the aforementioned I/O amplification issues, most B^+-tree-based systems incorporate a large *buffer pool* to temporarily hold updated data in memory and delay costly flush operations as long as possible. However, since each in-memory page is strictly mapped to a corresponding data page on storage, it requires significant space to accommodate dirty data pages in memory, even if only a few records are updated. As time elapses, much of the buffer pool space is wasted on buffering dirty pages, rather than effectively serving as a cache for speeding up data retrievals. The limited buffer pool resources are thus wasted, resulting in poor performance.

Issue #3: Logging dilemma. Buffering dirty data in memory poses another issue, data persistency. In order to ensure data integrity in the event of system failures, B^+-tree based systems temporarily store each updated record in a *change log* rather than solely in the volatile buffer pool to avoid costly flush operations that are required to persist data immediately to storage [26, 31, 43, 51, 58, 71]. Although it converts "in-place updates" to more efficient "out-of-place appending", upon system failures, the change log must be replayed to recover the lost data. An undesirable result is that the log, besides occupying significant storage space, also lengthens the recovery period, resulting in long system downtime and service outages. In order to limit the log size, periodic *checkpointing* is necessary to enforce the flush of dirty data pages, so that the corresponding records in the log space can be reclaimed promptly [26, 43, 51]. On the other hand, frequent checkpointing incurs many small and random I/Os, negatively impacting system performance. Eventually it creates a difficult trade-off between runtime performance, buffer pool and storage space usage, and recovery speed, resulting the *logging dilemma* in B^+-tree-based systems.

In this paper, we aim to revisit the design of the classic B^+-tree and fundamentally address the critical issues that have limited its effectiveness in the modern age. Our goal is to breathe new life into the decade-old B^+-tree structure by minimizing its I/O amplification, optimizing its use of the buffer pool and storage space, and enabling near-instant recovery.

1.2 Our Solution

All of the aforementioned issues essentially stem from the fact that record data in B^+-tree must be updated in place, directly in the leaf node. Although each update is appended in the change log for data persistency in the event of system failures, such updates are "temporary" until they are properly indexed by the B^+-tree index structure. In other words, these in-log changes are not persistently locatable until they are reflected in the original data pages during the checkpointing process. Simply extending the log size only postpones the need for "read-modify-write" operations for in-place updates, and it exacerbates the need for a large buffer pool and results in an unbearably long recovery time.

In this paper, we propose a novel solution, called *per-page logging* based B^+-tree or pB^+-tree, to address the challenges in traditional B^+-tree structures. Our key idea is to create a "virtualized" data page for each leaf node, which consists of the original data page and a small append-only region, called *per-page log*. In-place updates to a data page are then converted to "in-place appending", with all changes recorded in the per-page log. As each virtual page is addressable by the B^+-tree index structure, all updates are effectively persistent, completely eliminating the need for a large change log and enabling near-instant recovery upon system failures.

This solution essentially breaks down the huge global log into many small per-page logs and disperses them to each data node. It leverages the indexing structure of the existing B^+-tree to make updates in the log quickly persistent and index-addressable. This approach transforms expensive in-place writes to much lower-cost appending operations while removing the need for in-memory buffering and a large change log. It ultimately eliminates the cause of checkpointing-induced performance issues and shortens recovery time, thus freeing us from the difficult logging dilemma. However, this solution also poses a critical problem—a huge storage capacity waste due to the pre-allocated per-page logs.

Our solution is made possible by the emerging computational storage drives (CSD), which offers transparent on-device compression capability [1, 18, 72, 73, 86]. This new hardware technology provides a virtualized storage space and transparently compresses data on the I/O path. It enables us to create a large, over-provisioned "virtual" page for each node without wasting physical storage on reserved space for partially filled per-page logs. This is essential to our solution, as it allows us to fully realize the benefits of our proposed approach at a minimal storage cost. We have implemented a fully functional prototype of pB^+-tree and conducted experiments on a commercially available CSD from ScaleFlux [73]. To demonstrate its effectiveness, we have compared its performance with a conventional B^+-tree-based system with an ARIES-style logging mechanism [58]. Our evaluation results show that per-page logging and our proposed solution can significantly improve the run-time performance by up to 625.6%, with minimal physical space overhead and near-instant recovery.

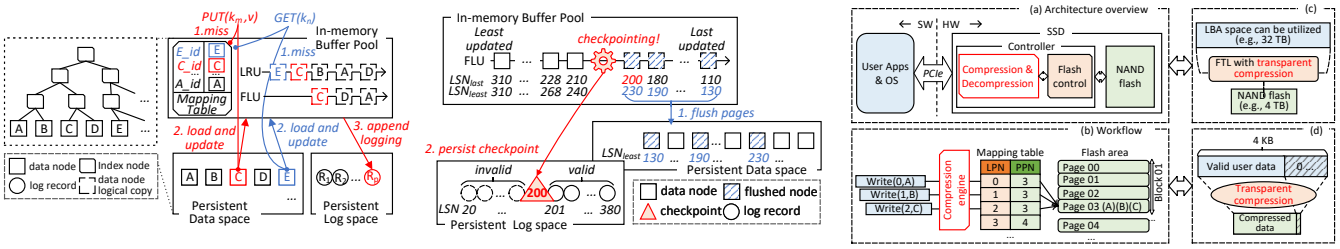
The rest of this paper is organized as follows. Section 2 presents the background. Section 3 and Section 4 introduce the proposed design solution and evaluation, respectively. Section 5 discusses the related work. Section 6 concludes the paper.

2 BACKGROUND

2.1 B^+-tree-based Storage Engine

B^+-tree-based storage engines are widely used and play a vital role in enterprise and customer applications [27, 46, 54, 57, 59, 61, 62, 65, 68, 75, 85]. Figure 1(a) illustrates the architecture of a typical B^+-tree, which consists of three major components: a persistent data store based on B^+-tree, an in-memory buffer pool, and a logging component [26, 31, 43, 51, 58, 71].

B^+-tree structure. A B^+-tree is an m-ary tree with a large number of children per node [26, 27]. It consists of three types of nodes: *root*, *internal*, and *leaf*. Data items are stored in leaf nodes, while internal nodes maintain the indexes of their children. By default, each node is allocated a fixed-sized page (e.g., 16KB) [59, 62]. To insert a record into the B^+-tree, the tree is first searched to find the leaf node that the new record should go into. If the node has sufficient space to accommodate the new record, the data is directly added to the node. Otherwise, the *split* operation is triggered, during which the original node is split into two nodes, and its data (data or index items) are evenly separated and put into the two new nodes. The indexes for the two nodes are inserted into their parent node. The split operation is recursively conducted for the involved sub-trees until all the nodes are under their size limit.



(a) B^+-tree-based system (b) Logging and checkpointing (c) CSD with built-in transparent compression

Figure 1: Illustrations of a B^+-tree-based system, logging and checkpointing, and CSD with built-in transparent compression.

In real-world deployment, B^+-tree nodes often suffer from significant under-utilization of the allocated space. Prior studies show that the average *fill factor*, i.e., the ratio of valid data to node size, is typically only around half of the node size [35, 41, 82, 83]. Our own experiments also show that the average fill factor is around 51.2% after a 90-minute random read/write workload. To address this space overhead, current enterprise B^+-tree solutions adopt software-based compression [9, 21, 28, 64, 86] or condensed data format [55, 77], but such approaches complicate the application design significantly and also introduce extra computation overhead. The new computational storage drive provides hardware support by offloading compression to the device without performance penalties. More details will be discussed in Section 2.3.

Memory buffer pool. B^+-tree-based storage engines often incorporate an in-memory *buffer pool* for two main purposes: caching frequently accessed pages and buffering updated (a.k.a., dirty) pages. Cached data pages are managed through an LRU (Least Recently Used) list. Dirty pages that have been modified in the *buffer pool* are tracked using two monotonically increasing *Log Sequence Numbers* (LSNs): LSN_{last} records the oldest revision of the page and LSN_{least} records the newest revision of the page. To keep track of these dirty pages, a FLU (Flush) list stores pointers to these pages based on their LSN_{last}. Once the *buffer pool* reaches a threshold, a flush operation is triggered to write the dirty pages from the FLU list to the storage in LSN order. It follows the write-ahead logging protocol [51, 59, 62, 71], which ensures that changes made to the pages are durable and recoverable in the event of system failures.

As illustrated in Figure 1(a), upon a write operation, the buffer pool is first queried to determine whether the corresponding data page is present in it or not. If true, the data item is updated directly in memory. Otherwise, the data page must first be loaded from storage into it before the update can be applied. Similarly, for read query operations, the buffer pool is also checked first. If the corresponding data page is found there, the data is returned. Otherwise, the data page needs to be loaded from storage into the buffer pool before the data is returned. The buffer pool serves write requests using an *in-place update* method, which requires loading the target data page from the slow persistent storage if it is not already present in the buffer pool. This operation, known as "read-modify-write", severely amplifies the I/O volume and introduces unpredictable, slow random I/Os, resulting in inefficiencies in B^+-tree-based systems, especially in handling intensive small writes.

2.2 Checkpoint-based Logging

The logging component is responsible for ensuring crash consistency by logging B^+-tree page updates and temporarily caching them in the *buffer pool*. This alleviates the performance impact of read-modify-write and random I/Os incurred by update operations. In the case of a system crash, B^+ can reconstruct all the not-yet-persisted dirty pages by replaying the updates in the change log. To reduce the crash recovery latency, a checkpointing process [43, 63] periodically flushes cached dirty data pages from the *buffer pool* to persistent storage according to the FLU and releases the corresponding space in the log.

Figure 1(b) illustrates an example of the checkpoint-based logging scheme. In the buffer pool, a FLU list keeps track of the updated data pages according to their corresponding LSN_{last}. Once the checkpointing process is triggered (e.g., due to too many dirty pages in the buffer pool or high log space occupation), a certain number of dirty pages (e.g., 100) at the end of the FLU list are selected and flushed from the buffer pool to storage. After this operation, the corresponding log records can be safely discarded. For instance, after the flush operation, if the LSN_{last} of the dirty page at the end of the FLU is 200, then the log records with LSNs smaller than 200 can be safely released. During recovery after a system crash, only log records after the latest checkpoint needs to be replayed to recover the involved data pages and reconstruct the database states, such as rebuilding the in-memory buffer pool. This process reduces the time required for recovery, as only a limited number of log records need to be processed.

Logging dilemma. Although checkpoint-based logging provides system crash consistency, the periodic flush operations it incurs can bring a heavy I/O burden and severely degrade system runtime performance. While reducing the checkpointing frequency or eliminating it completely may seem like an intuitive solution to decrease the involved I/O overhead and interference, it would result in occupying a huge (or infinite) log space and an unacceptably long recovery time upon system failures, since all the records in the change log must be replayed one by one. On the other hand, although frequent checkpointing operations reduce the log space usage as well as recovery time, the constantly happening checkpointing can also interfere with and degrade the performance of the foreground services. This creates a *logging dilemma*, for which there are currently no perfect solutions. This work shows that the emerging CSD with built-in transparent compression brings a unique opportunity to fundamentally address this logging dilemma.

2.3 In-Storage Transparent Compression

Generally speaking, CSD refers to any storage drives that can perform certain computational tasks beyond their core storage duties. In this work, we focus on one special type of CSD that provides

built-in transparent data compression capability. As shown in Figure 1(c.a), CSD controller chip integrates a hardware engine to compress and decompress each 4KB data block on the I/O path, and the *Flash Translation Layer* (FTL) is responsible for managing the storage of post-compression variable-length data blocks in NAND flash memory. Compared with host-side compression [9, 55, 64, 77, 86], such in-storage transparent compression completely relieves host CPU from compressing and decompressing data as well as managing the storage of post-compression variable-length data blocks.

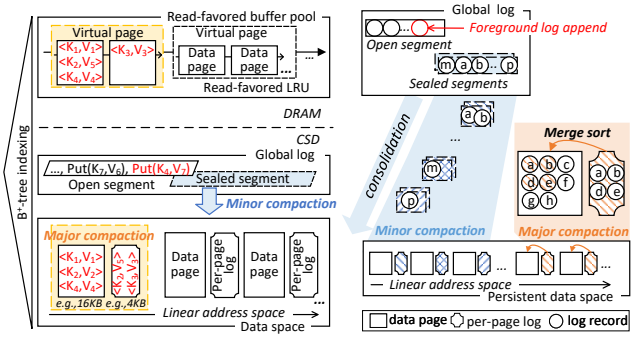
With a specifically designed ASIC as its hardware compression engine, the CSD with transparent compression achieves competitive performance, even compared with conventional commercial SSDs [40, 56, 70]. For example, the ScaleFlux CSD-3310 SSD, which is used in this work, achieves 1,020K IOPS under the sustained 70/30 random 4KB read/write workload with 2:1 compressible data, which is 191.4% and 240% higher than that of Samsung PM9A3 [70] and Micron 7450Pro [56] SSDs without in-storage compression, respectively. Additionally, since the intra-CSD per-4KB data compression is transparent to the host, CSD is 100% compliant with standard I/O interfaces, such as NVMe, which greatly simplifies its real-world deployment.

Figure 1(c.b) illustrates the data mapping inside CSD with transparent compression. Because NAND flash memory does not support in-place data update, any flash-based storage drives must internally maintain a mapping table to record the indirection mapping between *logical page number* (a.k.a., LPN) and *physical page number* (a.k.a., PPN). In traditional SSDs without built-in transparent compression [29, 30, 87], each incoming 4KB data block is directly written into one 4KB physical page, hence one LPN maps to one distinct PPN. In CSD with built-in transparent compression, each incoming 4KB data block is compressed before being written to NAND flash memory. As a result, multiple LPNs could map to one PPN, as illustrated in Figure 1(c.b): for the write operations updating logical pages 0, 1, and 2 with data A, B, and C, after compression, such three logical pages can be condensed into physical page 3 and then programmed to flash memory Block 01 for persistence.

CSD with built-in transparent compression offers two interesting properties. Firstly, as shown in Figure 1(c.c), it exposes a *virtualized* logical space that can be much larger than its physical storage capacity, conceptually similar to *thin provisioning* [23, 60, 67, 68]. This allows users to provision more logical storage capacity than that being physically available. Secondly, since repeated data patterns such as all-zero data are highly compressible, CSD allows users to pack *sparse* data inside each 4KB data block without sacrificing the physical storage cost, as shown in Figure 1(c.d). These two properties of CSD enable decoupling of user-perceived logical space from physical storage space. This is essential to our solution, as it allows data management software, such as the B^+-tree-based storage engines, to intentionally over-provision space for creating a *sparse* data structure in the logical storage space without sacrificing the physical storage capacity.

3 DESIGN

This section presents a novel design, called pB^+ tree, to fundamentally address the inherent problems with the traditional B^+ tree-based storage engine. Leveraging the vast, virtualized logical space provided by CSD, we maintain distributed per-page logs to achieve both high runtime performance and near-instant recovery. We desire to achieve three important goals:



(a) B^+-tree with per-page logging (b) An example of the per-page logging

Figure 2: An illustration of the architecture and workflow of the B^+-tree-based database system with per-page logging.

(1) **Runtime and recovery efficiency**. Compared with the traditional B^+-tree implementation, our solution should significantly reduce I/O amplifications under small-write intensive workloads and meanwhile ensure fast crash recovery.
(2) **Data persistence and correctness**. Our solution should not affect data integrity. Data persistence and correctness should remain identical to the traditional B^+-tree implementation.
(3) **High space utilization**. Our solution should not cause significant physical storage cost in comparison with traditional B^+-tree implementation.

We developed a set of design solutions to achieve the above-said design goals, and accordingly implemented and open-sourced a fully functional prototype based on real CSD hardware with built-in transparent compression. In the following sections, we will first introduce the overall design and then elaborate on each component.

3.1 Overview

As illustrated in Figure 2(a), our solution modifies the B^+-tree implementation from the on-disk storage and in-memory caching perspectives. While the data nodes are similar to those in the original B^+-tree, our proposed pB^+-tree maintains a *global log* and a set of distributed *per-page logs*. The global log is a small, append-only space used to batch small-sized and random updates, similar to the log in traditional implementation. The per-page logs, in contrast, are distributed over the storage space. Each B^+-tree page is attached with a per-page log, acting as a small and private log that keeps track of changes made to the records of this page. Thus, it can be regarded as a "delta" to the data page.

The memory part of pB^+ tree includes a *read-favored buffer pool* and an *in-memory global log index*. To minimize access latency and improve runtime performance, the *read-favored buffer pool* maintains a set of *virtual pages* that store copies of the valid data of hot data nodes in memory, based on their LRU sequence. Each virtual page is an in-memory copy of the most recent version of data records, which may reside in the data pages, the per-page logs, or the global log on persistent storage. The global log index maintains a small in-memory indexing structure for quick locating of target data in the global log.

3.2 Per-page Logging

To address the dilemma of logging, we leverage the large logical space provided by CSD to create a *sparse* data structure on storage, called *per-page logs*. Essentially, each per-page log is a small, over-provisioned logical space that contains updates made to its corresponding data page. As the per-page log is part of the data node, it represents a small in-place patch to the data records, making updates "index-addressable" by the B^+ tree index structure. As a result, the global log can be small, which allows us to decouple the trade-off between system performance and recovery time, fundamentally eliminating the root cause of the logging dilemma.

However, realizing the idea of per-page logging is non-trivial due to several challenges: (1) As per-page logs are distributed over storage space, directly appending each update to separate per-page logs individually would lead to massive small, random writes. As such operations reside on the critical path, slow random I/Os can directly impact foreground services and delay the response of requests. (2) As the delta to the data page, log records must be retrieved as well upon data requests, which could increase the latency and delay read operations. (3) Although CSD enables us to over-provision a virtualized logical space for accommodating per-page logs, its physical capacity is still limited. We must have a mechanism to ensure sufficient reserved space for continuous services.

• **Organization of Logs**. We propose a per-page logging design with a *two-phase logging* approach. In the first phase, we ingest incoming log writes in an append-only way to a small-sized *global log*. This avoids the immediate random writes to the distributed per-page logs, which would have a significant impact on foreground workloads. In the second phase, log records in the global log are asynchronously applied to the corresponding per-page logs in the background, moving I/Os from the critical path and leveraging the CSD's high I/O bandwidth.

Per-page logs are allocated in the granularity of pages. As shown in Figure 2(a), for each data node in the B^+-tree structure, pB^+-tree allocates a contiguous logical space consisting of two parts: the data page and a reserved space for the associated per-page log. Each data node is allocated with a 16KB data page and a 4KB per-page log, which together form a 20KB *virtual data page*. This design brings two benefits. First, by maintaining the per-page log space immediately after the data page, we do not need to maintain the extra mapping between them. Second, when accessing a data node, both its data page and log page can be read from storage together in one single I/O operation, which reduces the overhead. Per-page logging offers two important advantages: (1) By appending updates within its per-page log, read-modify-write operations now can be minimized. The per-page log accumulates updates as deltas to a data page, effectively reducing I/O amplification. (2) Benefiting from CSD's transparent compression, even partially filled per-page logs with only a few updates can be efficiently compressed, occupying a small amount of physical storage capacity. In other words, while the distributed per-page logs create a logically sparse data layout on CSD, physical storage space is not wasted.

Global log serves as a small, shared log space for all data nodes, providing a "staging area" to temporarily accumulate update operations and quickly persist the changes to storage. The updates in the global log are periodically flushed to their corresponding per-page logs in the background, which will be explained later.

The global log consists of an *open segment* and multiple *sealed segments*. An open segment is an active space that accommodates incoming updates. Once an open segment is filled, it becomes an immutable, "sealed" segment and waits to be written to the per-page logs. An in-memory index table is maintained for fast lookup in an open segment, and can be implemented as a hash-based key-value mapping table or a small B^+-tree. After a sealed segment is applied to per-pages logs, its in-memory index can be reclaimed.

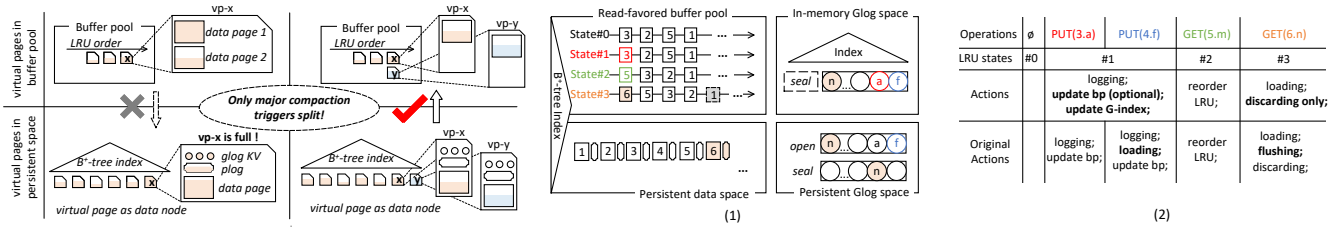
The global log brings two benefits. First, as logging is append-only, it only involves sequential writes, as opposed to random writes that would be incurred if we directly write to per-page logs. Second, the global log decouples the foreground runtime performance from per-page logging. A write operation can commit quickly and return once its operation log is appended to the global log, and the updates are persisted to the per-page logs later in the background. This effectively shortens the critical path for writes, allowing the log records to be asynchronously applied to per-page logs through parallel and batch I/Os with high bandwidth supported by CSD.

• **Log Space Management**. Two compaction processes, minor and major compaction, are responsible for log space management.

Minor compaction. Once an open segment in the global log becomes full, it transitions into a sealed segment and waits to be applied to the per-page logs. This process, called *minor compaction*, reclaims space from the global log, as illustrated in Figure 2(b). Since a sealed segment may contain multiple log records that belong to the same per-page log, applying the log records one by one to their per-page logs may incur redundant and unnecessary I/Os. Thus we conduct the compaction process in a batched manner. Specifically, we (1) scan the sealed segment and retrieve all the involved per-page logs from storage, (2) apply the log records to update the involved per-page logs in memory, and (3) flush the updated per-page logs back to storage. This batched processing exploits the parallelism of the CSD. In pB^+-tree, there may exist multiple seal segments, and their compaction processes can be performed in parallel.

Major compaction. When the per-page log becomes full, a *major compaction* process is triggered to merge the logged updates (deltas) to the corresponding data page, and the log space is reclaimed for accommodating future updates. As shown in Figure 2(b), once a per-page log exceeds its size limit (e.g., 4 KB), major compaction is initiated to (1) load both the log page and data page into memory in one I/O and merge-sort the key-value pairs, (2) update the data page with the merge-sorted results, and (3) reset the per-page log as all-zeros to reclaim the space. If the compacted data set exceeds the size limit of the data page, such as 16 KB in our example, a *split* operation is triggered to divide it into two data pages, and the tree's indexing structure (i.e., pB^+-tree's internal nodes) is updated accordingly.

To configure the frequency of major compaction, we can use a per-page log *fill factor*. By default, the fill factor is initially set as 1.0, meaning that the major compaction is triggered only when a per-page log is completely full. During runtime, we periodically measure the workload's *read ratio* within a time window (e.g., 10 minutes). When the read ratio reaches 80%, 90%, and 100%, we decrease the fill factor to 0.5, 0.25, and 0, respectively. The rationale behind this is that read-intensive workloads tend to benefit less



(a) Virtual-page-based B^+-tree structural operation

(b) Read-favored buffer pool management

Figure 3: Illustrations of (a) the virtual-page-based B^+-tree structural operation and (b) the read-favored buffer pool.

from per-page logs so we can do compaction earlier to save space and reduce data access overhead.

Handling updates and queries. The pB^+-tree with per-page logging is optimized for both updates and queries. For an incoming update operation, the system first checks the in-memory buffer pool to determine whether its data page is present or not. If yes, the update is appended to the open segment of the global log, and the in-memory data page is also updated correspondingly. Otherwise, the update is directly appended to the open segment of the global log without reading the data page from storage. This process, which we call *blind update*, differs from the original B^+-tree design, as it avoids the costly read-modify-write updates and consumes limited buffer pool space. Delete operations are handled as a special type of updates. When deleting a key, a new record with the key and a deletion flag as the value is created and inserted into the tree with an update operation. The space occupied by the deleted record is reclaimed during compaction operations.

For a query, the system first checks whether its data page exists in the buffer pool or not. If yes, the required data is directly returned. Otherwise, we load the valid data of this page from storage. In pB^+-tree, the "valid" data of one node page is the aggregated result of data in the data page and applied updates from the per-page log and the global log. Updates found in the global log are considered more recent than those in the per-page log, which, in turn, are more recent than the data records in the data page. Hence, it first loads the data page into memory and then consolidates the data from the logs as a *virtual page*, and then returns the data.

The correctness of concurrent updates and queries is guaranteed by the exclusive execution of operations during compaction and the strictly defined access order. Instead of blocking the entire compaction operation, only the compaction results (e.g., the updated global log, per-page log, and data page) are exclusively applied, during which accesses to the virtual page is lock-protected. In addition, the access flow follows the same order as the compaction operations (i.e., from the global log, per-page log, to the destination data page one by one, sequentially.). Thus, the concurrent updates and queries are guaranteed to retrieve the correct version of records.

In the event of a system crash, our per-page logging approach allows for a much faster recovery process compared to traditional designs. Instead of replaying a large number of log records (e.g., tens of GBs) to rebuild the entire buffer pool, we only need to reconstruct the small in-memory index for the open segment of the global log. This is possible because the majority of updates are already made index-addressable in the distributed per-page logs. As a result, we can achieve near-instant recovery by reconstructing the index for a small volume of log records (e.g., 100MB), enabling the system to be restarted immediately to handle incoming requests.

3.3 Virtual Page

The original B^+-tree design adopts a strict "one-to-one" mapping, where each in-memory data page is mapped to a data node in the tree, which has a corresponding on-storage copy of the same size. This ensures that any structural changes to the tree, such as a node split, can be equally reflected in both copies in a consistent way.

Such a design, however, cannot be directly applied in per-page logging, because the assumed one-to-one mapping is invalid. In pB^+ tree, the content of a data node is a result of aggregating the data records in the data page and the updates from the per-page log and the global log. As a result, the in-memory copy could exceed the size of the data page on storage, while a node split is not needed since the indexing is still valid. We need a new approach to handle page management in the buffer pool.

Virtual page. In pB^+-tree, we use *virtual page* as the basic caching unit for buffer pool management. A virtual page is a logical unit that is uniquely indexed by a data node in the B^+-tree structure. As illustrated in Figure 3(a), each virtual page (denoted as vp) maps to a B^+-tree data node and has an in-memory version and an on-storage version. The two versions are ensured identical.

The on-storage version of a virtual page is a logical group that consists of data records in the data page, updates in the per-page log, and unapplied record updates in the global log; The in-memory version is the aggregated result cached in the buffer pool, being organized in a list of 4KB pages in memory. If a memory page is filled, we do not directly split the data page as in the traditional B^+ design. Instead, we simply create a new memory page and insert it into the list of the virtual page to continue accommodating incoming records. Node split only occurs during major compaction, when the data page can no longer hold new updates and the split operation must be initiated since the data can no longer be indexed by a single data node in the tree structure.

Figure 3(a) illustrates split operation with virtual pages. In this example, the virtual page vp_x becomes full during major compaction. As the data page cannot accommodate all the data, vp_x is split into two new virtual pages, vp_x and vp_y, respectively. The new version of vp_x and the newly created vp_y, are then reflected in the buffer pool. This virtual page split process can be performed iteratively until achieved at the root node. Since only major compaction can trigger the split operation, uncoordinated splits can be avoided.

To reduce the frequency of split operations as well as major compaction, we can selectively flush data pages from the buffer pool. When a virtual page is evicted from the buffer pool, we check its fill factor. If the current virtual page contains more than one

data page, meaning it is oversized, we can directly conduct the split operation based on the data records retained in memory, and then disable and reclaim the records in the per-page log and the global log for the affected data page in memory. This ensures that the in-memory and on-storage versions of a virtual page are always consistent, and that structural correctness is maintained.

3.4 Read-favored Buffer Pool

With per-page logging, read-modify-write operations are no longer necessary, which eliminates the need to load original data page into memory. This presents an opportunity for us to redesign the buffer pool to prioritize reads. We can cache only frequently accessed, or "hot", data pages in the buffer pool for efficient reads, while making all write operations bypass the buffer and directly persist to the global log (a.k.a. *blind update*). This read-favored design can improve system performance by reducing cache pollution with infrequently accessed data pages, and prioritizing reads over writes.

In our proposed design, the buffer pool only caches hot data pages for reads, while all write operations bypass the buffer. Only a read LRU list is maintained to track the in-memory virtual pages. Whenever a virtual page is retrieved (accessed) from either the buffer pool or the persistent storage, it is inserted into the head of the linked list to maintain the LRU order. For the purpose of better illustration, Figure 3(b) shows an example with detailed LRU state transitions when serving a set of operations. We compare the behavior of the system with the ones without per-page logging or *read-favored buffer pool*. Initially, the LRU list is in State #0, as shown in Figure 3(b.1), and the four data pages (page ID 1, 2, 3, 5) are organized according to their LRU order, 3, 2, 5, and 1, respectively.

When we execute `PUT(3.a, value)` to insert a new key *a* into data page 3, the system follows these steps: (1) The record is first persisted in the global log space and the in-memory index is updated accordingly; (2) Since node page 3 is already in the buffer pool, it is directly updated and the LRU list remains unchanged.

For `PUT(4.f, value)`, there is no memory copy of the node page in the buffer pool, so we conduct a "blind update" by persisting the log record for key f in the global log and updating the in-memory index. The state of the LRU list remains unchanged and is still at State #1. In contrast, the traditional buffer pool management would need to load data page 4 for f into the buffer pool.

With per-page logging and the read-favored buffer pool, write operations can be performed without triggering random disk reads and flush operations when a buffer miss happens. As shown in Figure 3(b.2), such a blind update approach brings much better write performance, since random I/Os are eliminated from the write path. For read operations, the read-favored LRU list is still updated correspondingly. For example, on the third operation `GET(5.m)` that retrieves key *m* from node page 5, since the node page is already in the *read-favored buffer pool*, value can be directly fetched without storage access. The LRU list is updated by reinserting page 5 at the head of the list (State #2), similar to the original buffer pool design.

For the last operation, `GET(6.n)`, the node page is not present in the buffer pool and must be fetched from storage. Once the page has been loaded, it is inserted into the head of the LRU list, transitioning the list to State #3. Assuming the size threshold of the buffer pool is 4, the least recently used page, which is node page 1 in this case, would be evicted to make room for the newly fetched page.

Another benefit of the read-favored buffer pool design is the elimination of eviction costs. During the LRU eviction, the *read-favored buffer pool* can directly discard the in-memory copy without any flush operations. This is because any evicted page already has a persistent copy on storage, which can be found by aggregating records in the global log, the per-page log, and its data page, and the on-storage copy is guaranteed consistent with the in-memory copy. In other words, the page is "clean". Therefore, flushing the page is unnecessary, which saves the system from performing any random writes during the eviction.

In summary, with the read-favored buffer pool design, write operations are not buffered in the buffer pool, which saves the limited buffer pool space to better serve read operations. Additionally, since all data pages are clean, they can be safely evicted without flushing data back to storage, thus avoiding random I/Os and reducing the eviction overhead.

4 EVALUATION

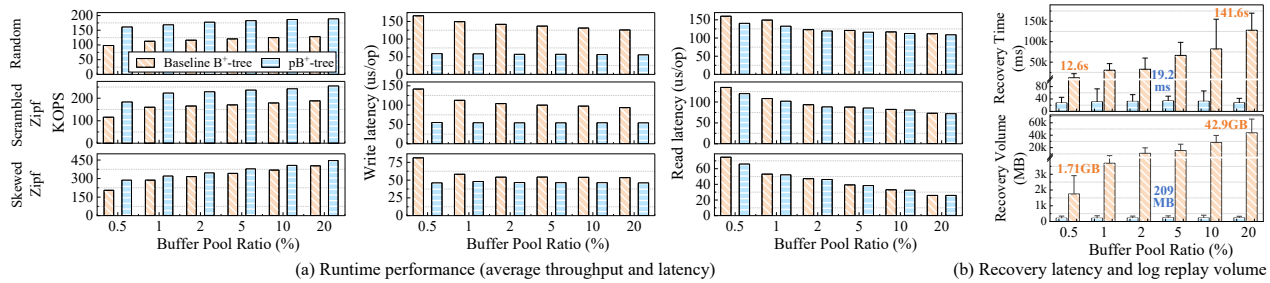
To evaluate *pB$^+$-tree*, we have developed a fully functional prototype that incorporates the proposed per-page logging, virtual pages, and read-favored buffer pool. For comparison, we have also implemented a baseline B^+-tree that uses conventional buffer pool management with ARIES-based write-ahead logging [58], denoted as *baseline*. We also compare with WiredTiger [59], a production-grade key-value engine based on B^+-tree data structure. All three prototypes provide standard key-value interfaces, such as `PUT()` and `GET()`. We conduct our evaluations on the latest commercially available ScaleFlux's CSD with built-in transparent compression [73].

4.1 Experimental Setup

Our experiments are conducted on a server equipped with a 12-core 2.1GHz Intel Xeon Silver 4310 CPU, 128GB DDR4 DRAM, and a 3.82TB ScaleFlux CSD-3310 [72, 73]. The server runs Ubuntu 20.04 LTS with Linux Kernel 5.15 and Ext4 file system. We use the *io_uring* [38], a Linux kernel system call interface to perform I/O operations, such as `write()` and `read()` in an asynchronous manner for both the baseline and the pB^+-tree prototype.

The ScaleFlux's CSD-3310 integrates a hardware-based zlib compression engine that compresses each 4KB block on the internal I/O path, which is transparent to the host. The engine has a compression and decompression latency of around 5μs, which is over 10× shorter than the read latency (>50μs) and much less than write latency (>1ms) of typical TLC/QLC NAND flash memory. Being 100% compliant with NVMe standard, the drive operates with a PCIe Gen4 ×4 interface, achieving up to 7.2GB/s and 4.8GB/s sequential read and write bandwidth, respectively. It can perform 1450K/380K random 4KB read/write IOPS (I/O per second) over 100% LBA span. The CSD-3310 drive is already in volume production and has been deployed in data centers worldwide.

To evaluate the performance of our proposed pB^+-tree design solution and compare it with a baseline B^+-tree, we conduct experiments using Yahoo! Cloud Serving Benchmark (YCSB) [19]. Before running the benchmark, we first warm up both the baseline B^+-tree and pB^+-tree by sequentially populating each store with 100 GB of key-value pairs. The key-value pair size varies between 100 bytes and 200 bytes, with an average of 120 bytes. For each key-value



(a) Runtime performance (average throughput and latency)

(b) Recovery latency and log replay volume

Figure 4: Comparison of (a) run-time and (b) recovery performance under different workloads.

pair, we generate the key based on YCSB's distribution to emulate real-world applications, and the value is filled with randomly generated data content. We evaluate the systems using different read/write ratios and configurations, with a fixed data set size of 100 GB for each test case. Specifically, we used YCSB's Random Distribution (denoted as *Random*), Scrambled Zipfian Distribution (denoted as *Scrambled Zipf*), and Skewed Zipfian Distribution (denoted as *Skewed Zipf*) to generate the keys.

4.2 Overall Performance

We first conduct experiments to demonstrate the overall performance and crash recovery efficiency of the pB^+-tree design. For both the *baseline* and *pB^+-tree*, we set the size of a B^+-tree node page as 16KB. All the non-leaf tree nodes are kept in memory, while the leaf nodes that contain all the key-value pairs are persistent on the CSD. The baseline's buffer pool management employs the original LRU and FLU lists for flushing, with the flush thresholds set as 90% and 70%, respectively. The buffer pool size is 1.5GB by default. For pB^+-tree, we set the per-page log size as 4KB and the size of the open segment in the global log space as 100 MB. The read-favored buffer pool of the pB^+-tree uses the LRU policy similar to the baseline, which size (plus the size of the global log index) is similar to the buffer pool size of the baseline.

We evaluate the performance of both the baseline and pB^+-tree using 16 concurrent clients under three workloads with mixed 50% read and 50% write operations. The impact of the number of clients and read/write ratios are analyzed in Section 4.3. Figure 4 shows the overall performance and recovery speed of the baseline and pB^+-tree as we increment the buffer pool ratios (i.e., the ratio of buffer pool size to the dataset size of 100GB) from 0.5% to 20%.

Overall performance. In Figure 4(a), we can observe that pB^+-tree outperforms the baseline in terms of run-time performance (throughput and read/write latency) across all workloads. The largest performance improvements are achieved under the workload with random distribution, where the pB^+-tree increases the throughput by up to 63.85% and reduces the write and read latencies by up to 64.73% and 12.3%, respectively. Under the Skewed Zipf workload, pB^+-tree also exhibits significant performance improvement, increasing the throughput by up to 39.54% and reducing the write and read latencies by up to 43.46% and 11.85%, respectively. The benefits are more moderate under the Scrambled Zipf workload, where pB^+-tree increases the throughput by 34.97% to 58.27%, and reduces the write and read latencies by 42.36% to 61.2% and 1.43% to 11.22%, respectively. The performance benefits of pB^+-tree can be attributed to two main factors. First, the *per-page logging* approach, which performs blind updates, shortens the write path and reduces system latency. Second, pB^+-tree induces far fewer disk I/Os for minor and major compaction compared to the baseline's read-modify-write approach.

As shown in Figure 4(a), the relative advantage of pB^+-tree decreases as the buffer pool size increases. This is because a larger buffer pool allows the baseline to accumulate more reads and writes, while pB^+-tree chooses to bypass the buffer pool for writes, and the consolidations are conducted by the global log, which has much less space than the buffer pool. Additionally, pB^+-tree shows less performance improvement under workloads with better localities, such as the Skewed Zipf. However, even with a large buffer pool, pB^+-tree still substantially outperforms the baseline under all workloads, when the size ratio increases to 20% of the dataset size.

Crash recovery. An important advantage of pB^+-tree is its "instant recovery" enabled by the per-page logging. With the baseline, while a large buffer pool can improve its run-time performance, it leads to a long recovery time as a large number of log records need to be replayed when the system crashes. With the per-page logging of pB^+-tree, only the log entries in the small global log need to be replayed during recovery, making the recovery process much faster and more efficient. Figure 4(b) shows the recovery time comparison between the baseline and pB^+-tree with buffer pool sizes varying from 0.5% to 20%. As the buffer pool size increases, the recovery time of the baseline B^+-tree also increases significantly, from 12.6s to 141.6s on average of ten arbitrary system aborts and restarts. This is because the larger buffer pool size holds more key-value pairs in memory, resulting in larger log space (increasing from 1.71GB to 42.9GB), which needs to be scanned and replayed during recovery, causing longer recovery latency.

In contrast to traditional logging approaches where recovery time is directly proportional to the buffer pool size, pB^+-tree's per-page logging design decouples recovery concerns from runtime performance. During recovery, pB^+-tree only needs to reconstruct the in-memory index for the global log, which contains a small volume of log records (around 209MB on average). After the index is reconstructed, the system can immediately start serving incoming requests. On average, pB^+-tree achieves a 19.2 ms recovery latency, which is far less than that of the baseline. These results demonstrate that pB^+-tree achieves near-instant recovery regardless of buffer pool sizes and outperforms the baseline in terms of recovery latency.

4.3 Analysis on Per-page Logging

This section analyzes the per-page logging design and the effect of each component of pB^+-tree. Specifically, we conduct several evaluations to explore how pB^+-tree's better bandwidth utilization

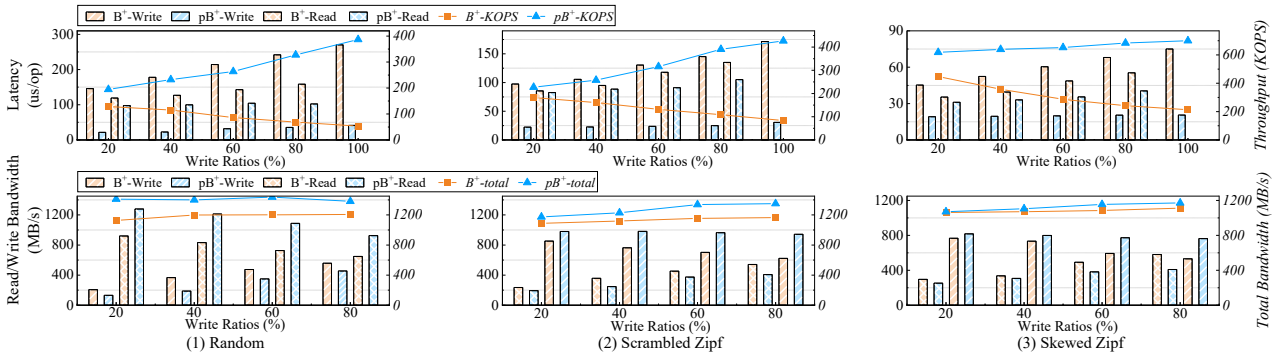


Figure 5: Performance and bandwidth utilization with write ratios from 20% to 100% under different workloads.

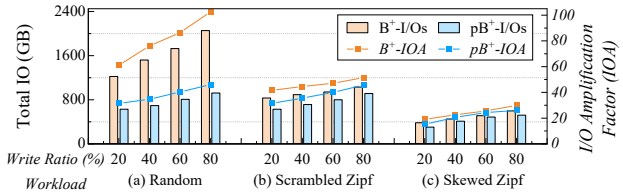


Figure 6: Total I/Os and I/O amplification (IOA) with write ratios from 20% to 80% under different workloads.

and reduced I/O amplification contribute to the improved performance. We also evaluate the concurrent performance of pB^+-tree with various numbers of client threads under different workloads, and examine the space occupation of per-page logs with the CSD. Finally, we analyze how the fill factors affect the system performance with particular workloads, providing a comprehensive understanding of pB^+-tree's design and its implications for database systems.

Per-page logging. Figure 5 shows the performance and I/O bandwidth utilization of pB^+-tree and the baseline under different read/write ratios, revealing the significant benefits brought by the per-page logging. We can see that pB^+-tree outperforms the baseline in terms of throughput (KOPS) and average latency in all cases.

Firstly, as the write ratio increases, pB^+-tree demonstrates increasing overall throughput, while the baseline's overall throughput decreases. The throughput improvements of pB^+-tree over the baseline range from 51.4% to 625.6%, 24.6% to 410.3%, and 38% to 228.2% for the workloads of Random, Scrambled Zipf, and Skewed Zipf, respectively. Secondly, the latency of pB^+-tree remains stable as the write ratio increases, whereas the latency of the baseline continues to increase. In particular, the write latency reductions range from 85.2% to 87.4%, 77.2% to 82.2%, and 57.7% to 72.7%, for the three workloads respectively; and the read latency reductions range from 18.3% to 35.5%, 12.2% to 26.9%, and 3.3% to 22.2%, respectively. These results demonstrate that per-page logging significantly improves the performance and reduces the latency of the database system, particularly when the workload involves a higher write ratio.

The lower part of Figure 5 provides further insights into the reasons why pB^+-tree outperforms the baseline. First, the per-page logging of pB^+-tree allows blind updates and separates foreground service from background I/O operations. This leads to better utilization of the high bandwidth provided by the CSD, resulting in better overall throughput and lower write latency compared to the baseline. On average, pB^+-tree demonstrates 19%, 12.3%, and 4% higher bandwidth utilization than the baseline for the Random, Scrambled Zipf, and Skewed Zipf workloads, respectively.

Second, the per-page logging design of pB^+-tree results in less I/O amplification compared to the baseline. The severe I/O amplification of the baseline (especially write amplification) leads to significant read/write contention, requiring the baseline to allocate more bandwidth for writes (53%, 30.2%, and 17% on average over pB^+-tree for the three workloads). As a result, fewer bandwidths (30.6%, 24.1%, and 17% on average less than pB^+-tree for the three workloads) can be provided for reads, leading to longer read latencies in the baseline as compared to pB^+-tree.

In Figure 6, we further analyze the total I/Os and I/O amplification of pB^+-tree and the baseline. The I/O amplification is calculated as the ratio of CSD's total physical read/write to the host's issued total read/write. The results show that the baseline generates more I/Os than pB^+-tree across all three workloads, with I/Os that are 112.6%, 21.9%, and 13.7% higher than pB^+-tree, respectively. The I/O amplification of the baseline over pB^+-tree also follows the same trend. It shows that per-page logging and the read-favored buffer pool significantly reduce I/O amplification, which contributes to the improved system performance of pB^+-tree.

Concurrency performance. Figure7 shows the concurrency performance of pB^+-tree, in terms of the system throughput and average latency under different numbers of client threads. For system throughput, as the number of client threads increases, the improvements of pB^+-tree over the baseline increases from 116.4% to 241.8%, 127.6% to 137.1%, and 97.6% to 115% for the three workloads, respectively. As for the latency, with the increasing concurrency, the average latencies of pB^+-tree remains stable while the baseline shows an increasing trend. Specifically, the pB^+-tree's average write latencies are 81.7% to 85.2%, 78.9% to 83.4% and 55.9% to 67.1% shorter than the baseline's for the three workloads, respectively; and the average read latencies are 4.1% to 27.2%, 4.9% to 36.7%, 1.2% to 27% shorter for the three workloads, respectively.

The improvements come from two aspects. (1) pB^+-tree's blind update can better take advantage of the CSD's high bandwidth. Consolidated I/Os accumulated by blind updates and compaction operations can build a deeper NVMe queue, which in turn contributes to higher system throughput with more concurrent requests. (2) pB^+-tree can better control I/O amplification. Taking advantage of blind update and the read-favored buffer pool design, the potential I/O contention can be reduced, so pB^+-tree exhibits shorter latency.

Space consumption. We further evaluate the space usage of the pB^+-tree and the baseline under different dataset sizes, ranging from 50GB, 100GB to 200GB. In each experiment, the dataset is randomly

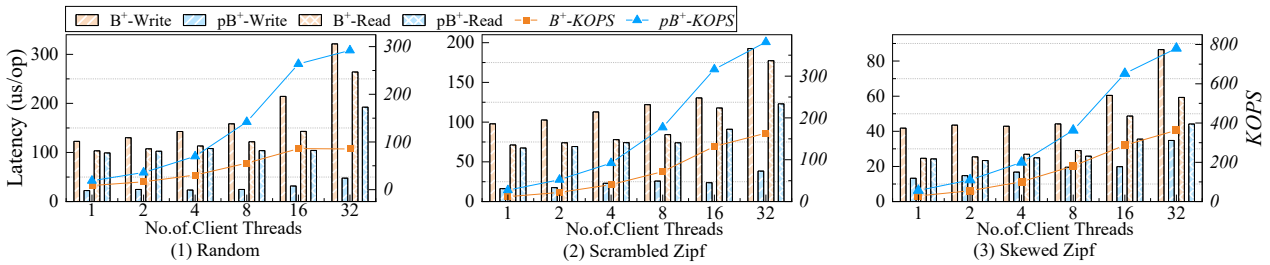


Figure 7: Throughput and latency under different workloads with the number of clients from 1 to 32.

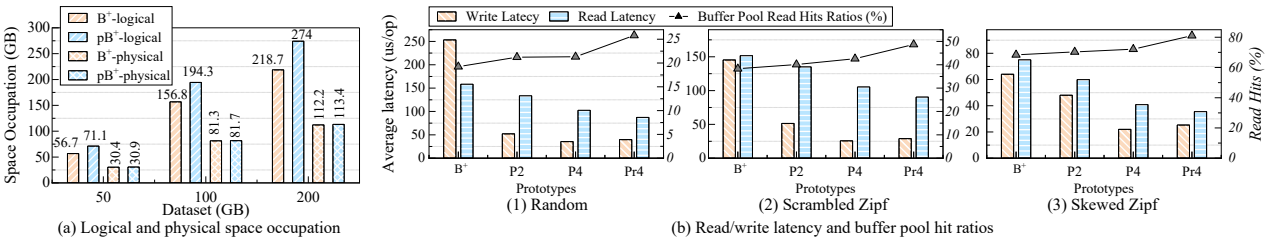


Figure 8: Comparison of (a) logical/physical space utilization and (b) performance and buffer pool hit ratio comparison between the baseline and pB^+-tree with different configurations.

accessed with a mix of 50% read and 50% write key-value operations. We set the size of the per-page log for the pB^+-tree as 4KB, and the size of the data page for the pB^+-tree and the baseline as 16KB. Figure 8 (a) compares the total storage usage in terms of logical storage usage (i.e., before in-storage compression) and physical flash memory space usage (i.e., after in-storage compression).

In terms of logical space usage, as the per-page logs are sparsely managed with each of them being partially filled, pB^+-tree requires more logical space than the baseline. Specifically, for the 50GB, 100GB, and 200GB datasets, pB^+-tree consumes 25.4%, 23.9% and 25.3% more space than the baseline, respectively. However, in terms of physical flash memory space, pB^+-tree consumes nearly identical space to that of the baseline, with only a slight increase. For 50GB, 100GB, and 200GB datasets, the extra physical space occupied by pB^+-tree over the baseline is only 1.6%, 0.5% and 1.1%, respectively. It is because the CSD with built-in transparent compression can condense the sparse per-page logs before writing to the physical space. Since most per-page logs are only partially filled (e.g., only 1KB space stores actual data while the rest of the space is filled by zeros), the over-provisioned logical space requires much smaller real physical space after transparent compression inside the CSD.

Fill factor. As mentioned in Section 3, the per-page log fill factor can also affect the performance of pB^+-tree and should be configured according to the runtime workload characteristics. For example, under read-dominant workloads, read performance can be improved if the valid data in the global log and per-page log are timely consolidated to its data pages, for which we should reduce the per-page log fill factor. On the other hand, under write-intensive workloads, reducing storage write amplification is more important, for which we may increase the per-page log fill factor.

In this work, we adopt a simple heuristic to adjust the per-page log fill factor. We adaptively tune the fill factor of a 4KB per-page log with a read-favored policy (i.e., Pr4) according to the read ratios of the incoming requests during a sliding time window. For example, when a read-dominant workload is detected (e.g., requests are all read in one minute), we reduce the fill factor of per-page logs by

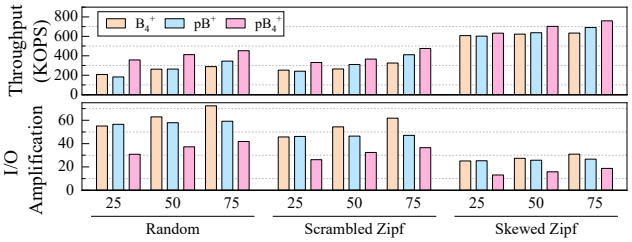


Figure 9: Performance and I/O amplification under alternative B^+-tree configurations.

half, meaning that major compaction would be triggered more aggressively by reclaiming a per-page log.

We have implemented three different versions of the pB^+-tree for comparison: one with 4KB per-page log (referred to as P4), one with 2KB per-page log (referred to as P2), and one with 4KB per-page log and dynamically adjusted fill factor (referred to as Pr4). The results in Figure 8(b) show that dynamically adjusting the fill factor for 4KB per-page logs can reduce the read latency of Pr4 by 40.44% to 52.67%, 33.06% to 40.73%, and 12.91% to 14.67%, compared to the baseline, P2 and P4 for the three workloads, respectively. The read-hit ratios (e.g., ratios of requests cached by the buffer pool) also show that Pr4 is more efficient than the others.

4.4 Trade-offs of Per-page Logging

• **Comparing to alternative B^+-trees**. We further evaluate the performance of the proposed pB^+-tree with varying data node sizes. In Figure 9, B_4^+, pB_4^+ and pB^+ refer to the baseline B^+-tree, pB^+-tree with 4KB data page, and pB^+-tree with 16KB data page.

As shown, pB_4^+ outperforms B_4^+ in all cases in terms of the system throughput and I/O amplification. The highest improvements of pB_4^+ over B_4^+ are achieved under the Random workload (up to 72.0% and 43.9% for system throughput increment and I/O amplification reduction); the least improvements are observed under the Skewed Zipf workload (up to 19.6% and 37.8%, respectively). The benefits mainly come from the blind update and I/O reduction enabled by per-page logging and read-favored buffer pool designs, as described in Section 4.3.

With read-intensive workloads (25% write ratio), pB^+ with 16KB data page shows a slight performance loss of 4.05% compared with B_4^+. The reason is that for a query that is not hit in the buffer pool, pB^+ incurs 20KB I/Os (16KB data and 4KB log), while B_4^+ only incurs 4KB I/Os. For the same reason, compared with B_4^+, pB^+ triggers 2.62% higher I/O amplification under read-intensive workloads, which are mainly from foreground I/Os for serving read operations. As the write ratio increases to 50% and 75%, we can see that pB^+ outperforms B_4^+ in terms of system throughput and I/O amplification across all the workloads. The benefit mainly comes from the blind update design of pB^+ which eliminates the I/O-intensive checkpointing process (i.e., background I/Os) by compaction operations as described in Section 4.2. It is also worth noting that although adopting a small page size (4KB) seemingly alleviates the involved I/Os, it leads to multiple negative effects, such as the large index size and increased lock contention in buffer pool management, which makes it a less preferable configuration in many databases [44, 62, 65, 76].

Table 1: Performance of pB^+-trees with per-page logs and data pages stored on separate devices.

Prototype	Throughput (KOPS)	Average Write Latency (us/op)	Average Read Latency (us/op)
pB^+-tree	287.4	25.41	101.3
(p)B^+-tree	277.8	21.14	107.7

• **Separating per-page log from data page**. To explore the feasibility of storing the log and data in separate storage devices for handling disk failures, we expand the standard design of pB^+-tree to a version storing per-page logs and data pages in two ScaleFlux CSD-3310 devices separately, denoted as (p)B^+-tree. As shown in Table 1, under the 50/50 Random read/write workload, the average read latency of (p)B^+-tree is 6.31% longer than that of pB^+-tree, since a read to the virtual page is now divided into two separate and synchronous I/O operations to two storage devices. However, (p)B^+-tree shows 16.8% write latency reduction. It is because pB^+-tree performs blind update for write operations, and the asynchronous and batched compaction can better utilize the doubled bandwidth (provided by two storage devices). The overall throughput of (p)B^+-tree is slightly (3.3%) lower than that of pB^+-tree.

• **Effects of global log**. We also evaluate the performance and I/O effects by varying the segment size of the global log. Based on the results, as the segment size increases from 50MB to 100MB, both throughput (from 147.6 KOPS to 197.7 KOPS) and total I/Os (from 202.5 GBs to 144.3 GBs) show significant improvements (reductions). However, when the segment size exceeds 100MB, the throughput and I/O show little further improvement (no more than 204.5KOPS and no less than 140.3GB, respectively). Meanwhile, as the segment size increases from 50MB to 500MB, the recovery time and volumes sharply grow from 4.4ms to 152.4ms and 74.7MB to 1024.3MB, correspondingly.

• **Effects of clients**. To saturate the bandwidth of the storage device and explore the potential interference between asynchronous threads, we initiate 64 to 256 clients to constantly perform random read/write operations to pB^+-tree with write ratios ranging from 80% to 100%. The results show that the performance stops climbing and begins to decrease with 128 clients under 90% write case, which is caused by the severe interference as the thread contention and storage bandwidth become system bottlenecks.

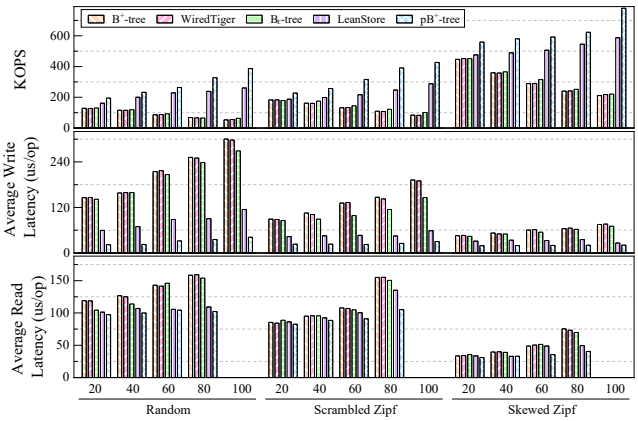


Figure 10: Performance comparison of the baseline B^+-tree, WiredTiger, $B\varepsilon$-tree, LeanStore and pB^+-tree.

• **Comparing to pB^+-tree without per-page log**. To illustrate the performance and I/O impact of per-page log, we further disable the 4KB per-page log of pB^+-tree with 16KB or 4KB data pages by configuring the size of all per-page logs to 0, which forces minor compaction to directly flush the records in the global log to data pages. We issue the Random mixed read/write workloads to compare the performance and I/O amplification of pB^+-tree with and without per-page log. The storage device is a 1TB Western Digital SN570 NVMe SSD, which has no compression capability.

After disabling per-page logs, pB^+-tree with 16KB data pages shows non-trivial throughput degradation (up to 36.3%) and I/O amplification (up to 31.8%) compared to pB^+-tree with 4KB per-page logs and 16KB data pages. It is because after disabling per-page logs, each minor compaction directly rewrites a 16KB data page, introducing severe write amplification. Although the global log decouples foreground services from background compaction, the amplified background compaction still causes interference to the foreground service operations and results in performance degradation. In addition, pB^+-tree with 4 KB per-page logs and 16KB data pages show 21.2% lower throughput and 30.1% higher I/O amplification compared to pB^+-tree with 4KB data pages and without per-page logs. It is because pB^+-tree with 4KB data pages introduces less read amplification than pB^+-tree with 4 KB per-page logs and 16KB data pages. However, considering that large data page size reduces index management overhead, allocating a small-size per-page log for each data page can take advantage of blind updates and reduce the involved I/O amplification, which is considered a more beneficial choice overall.

4.5 Comparison with State-of-the-arts

In this section, we compare the performance of pB^+-tree with two state-of-art B^+-tree-based systems, WiredTiger [59], LeanStore [2, 44] and $B\varepsilon$-tree [7]. We conduct experiments using a mixed workload lasting for one hour with varying read/write ratios for all the prototypes. Similar to prior experiments, the workloads are based on YCSB's Random, Scrambled Zipf, and Skewed Zipf distributions.

Figure 10 shows the results in terms of average throughput, write and read latency. Comparing the baseline B^+-tree with WiredTiger, we can find that our implementation of the baseline achieves robust performance similar to that of the production-level B^+-tree-based

storage engine. In addition, we can see that our pB^+-tree outperforms both the baseline B^+-tree and WiredTiger across the board. Specifically, pB^+-tree achieves improvements ranging from 25.13% to 625.59%, 57.73% to 86.22%, and 3.33% to 46.28% compared to the baseline, and 23.78% to 599.48%, 58.54% to 86.1%, and 2.02% to 32.29% compared to WiredTiger, in terms of throughput (KOPS), average write and read latency (μs/op) under the three workloads.

Moreover, comparing with the state-of-art optimizations for B^+-tree-based implementations, $B\varepsilon$-tree and LeanStore, our pB^+-tree also shows substantial performance improvements. Specifically, pB^+-tree outperforms $B\varepsilon$-tree by 23.4% to 516.9%, 56.26% to 85.86% and 2.81% to 27.65%, and outperforms LeanStore by 14.25% to 58.34%, 21.42% to 64.65% and 0.02% to 25.3% in terms of throughput, average write and read latency, respectively. We also measure the total I/Os issued by pB^+-tree, LeanStore and $B\varepsilon$-tree during the period of ingesting 20GB random writes. The results in Table 2 show that $B\varepsilon$-tree and LeanStore issue 81.35% and 71.63% more I/Os compared to pB^+-tree, and the corresponding write amplifications of LeanStore and $B\varepsilon$-tree are 1.81x and 1.72x of pB^+-tree.

Table 2: Physical I/Os for ingesting 20GB random writes.

Prototype	Physical I/Os (GB)	Physical Write Amplification
pB^+-tree	**467.1**	**23.36**
LeanStore	847.1	42.36
$B\varepsilon$-tree	801.7	40.09

This experiment well demonstrates that the pB^+-tree design achieves substantial performance improvement even compared with the production-level implementation of the traditional B^+-tree and the state-of-art B-tree optimizations.

5 RELATED WORK

B^+-tree-based data systems have been heavily studied. Prior works have been conducted on various aspects to improve the database performance, storage efficiency, and data reliability, etc.

Structural optimization for B^+-tree. Many structure-related techniques have been proposed to optimize the efficiency and performance of B^+-tree-based database systems [10, 12, 16, 26, 27, 37, 46, 47, 50, 68, 80, 81]. Bw-tree [46, 81] is a representative design adapting B^+-tree to modern multi-core CPU architecture with fast SSDs. The Bw-tree organizes each B^+-tree logical node as "base plus deltas", which enables latch-free operations for better utilizing multi-core CPUs, and reduces write amplification and write stalls by persisting pages and deltas with a log-structured store. However, their solution still follows the "read-modify-write" principle and requires extra indirect mappings. Many works also optimize B^+-tree for persistent memory [5, 20, 22, 33, 37, 45]. For instance, SLM-DB [37] proposes to leverage persistent memory by combining the indexing efficiency of the B^+-tree and the write efficiency of LSM-tree [13, 48, 66, 78, 84], such that both the fast lookup/ingestion can be realized. These proposed techniques are orthogonal to our per-page logging, and thus can be applied together to further improve the efficiency of the B^+-tree-based database system.

Logging optimization for B^+-tree. ARIES [58] has been the de facto standard for logging and recovery in B^+-tree-based and many other systems. It provides a mechanism including write-ahead logging and checkpointing to enable crash recovery as well as native support for indexes and space management. Many works have been proposed to optimize the efficiency of ARIES [4, 24–26, 31, 39, 43, 51, 52, 74]. For example, Haubenschild *et al.* [31] propose several techniques, such as parallel logging, optimized commit protocol and checkpointing mechanism, to exploit the potential of ARIES with fast storage devices and many-core architecture. Lee *et al.* [43] and Magalhaes *et al.* [51] propose to maintain indexes for log records and conduct the checkpointing for indexed logs to achieve instant recovery for memory-optimized cloud-native B^+-tree databases and purely in-memory B^+-tree databases, respectively. These solutions, which aim to boost either the logging or recovery process, are specially tailored to particular components. Unlike these prior works, our per-page logging divides and distributes the log records in a sparse data layout on storage, which enhances the B^+-tree structure to naturally enable blind update and instant recovery, fundamentally addressing the logging inefficiency for B^+-tree-based database systems with ARIES logging.

B^+-tree on CSD. Qiao *et al.* [68] show that CSD with built-in transparent compression is beneficial for B^+-trees. Such devices can expose a virtualized logical address space that is much larger than the actual physical storage capacity of the device, which is realized by implementing compression directly on device with the help of FTL. By leveraging the sparse logical address space, the I/O amplification issues inherent in the B^+-tree design can be mitigated with various techniques, such as page shadowing and optimized logging, etc. Other attempts [1, 14, 15, 18, 53], such as directly building B^+-trees on flash devices [1], also show the potentials of B^+-tree in many domains. Our per-page logging approach is built atop CSD with transparent compression. Our studies well demonstrate that with proper designs and assistance from new hardware, B^+-trees can efficiently serve modern workloads with high performance and low cost.

6 CONCLUSION

This paper presents a simple yet effective design that leverages the emerging CSD with built-in transparent compression to fundamentally resolve the logging dilemma of B^+-tree implementations. Besides improving data storage efficiency, a unique advantage of CSD is to allow data management software to deliberately create sparse data structures on storage without sacrificing its physical capacity. Leveraging this property, we propose a per-page logging based B^+-tree design to seamlessly decouple the crash recovery latency from the log size. Its effectiveness has been well demonstrated through extensive experiments and comparison on a commercially available CSD with built-in transparent compression.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and insightful comments. The work described in this paper was partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14219422 and GRF 14202123), Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055151), the National Science Foundation under Grants CNS-2006617, CCF-2210754, and CCF-2210755, the National Natural Science Foundation of China under Grant 62272271, and Key Research and Development Program of Shandong Province under Grant 2023CXPT002.

REFERENCES

[1] Ahn, Jung-Sang and Kang, Dongwon and Jung, Dawoon and Kim, Jin-Soo and Maeng, Seungryoul. 2012. μ^*-Tree: An Ordered Index Structure for NAND Flash Memory with Adaptive Page Layout Scheme. *IEEE Trans. Comput.* 62, 4 (2012), 784–797.

[2] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 6 (2023), 1426–1438. https://www.vldb.org/pvldb/vol16/p1426-alhomssi.pdf

[3] Alibaba. 2022. Alibaba Block Trace. https://github.com/alibaba/block-traces.

[4] Antonopoulos, Panagiotis and Byrne, Peter and Chen, Wayne and Diaconu, Cristian and Kodandaramaih, Raghavendra Thallam and Kodavalla, Hanuma and Purnananda, Prashanth and Radu, Adrian-Leonard and Ravella, Chaitanya Sreenivas and Venkataramanappa, Girish Mittur. 2019. Constant Time Recovery in Azure SQL Database. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2143–2154.

[5] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (2018), 553–565. https://doi.org/10.1145/3187009.3164147

[6] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martin Farach-Colton, and John D. Owens. 2019. Engineering a high-performance GPU B-Tree. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, New York, 145–157. https://doi.org/10.1145/3293883.3295706

[7] 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). https://www.usenix.org/publications/login/oct15/bender

[8] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, CA, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[9] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. http://www.vldb.org/pvldb/vol13/p2649-boncz.pdf

[10] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower bounds for external memory dictionaries. In *Annual ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM, Baltimore, 546–554. http://dl.acm.org/citation.cfm?id=644108.644201

[11] Cha, Hokeun and Nam, Moohyeon and Jin, Kibeom and Seo, Jiwon and Nam, Beomseok. 2020. B3-tree: Byte-addressable Binary B-tree for Persistent Memory. *ACM Transactions on Storage (TOS)* 16, 3 (2020), 1–27.

[12] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. 2021. Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine. *Proc. VLDB Endow.* 15, 1 (2021), 112–126. https://doi.org/10.14778/3485450.3485461

[13] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, CA, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[14] Xubin Chen, Ning Zheng, Shukun Xu, Yifan Qiao, Yang Liu, Jiangpeng Li, and Tong Zhang. 2021. KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression. In *International Workshop on Data Management on New Hardware (DaMoN)*. ACM, China, 3:1–3:10. https://doi.org/10.1145/3465998.3466004

[15] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, and Sheng Wang. 2022. CloudJump: Optimizing Cloud Databases for Cloud Storages. *Proc. VLDB Endow.* 15, 12 (2022), 3432–3444. https://www.vldb.org/pvldb/vol15/p3432-chen.pdf

[16] Chen, Youmin and Lu, Youyou and Fang, Kedong and Wang, Qing and Shu, Jiwu. 2020. uTree: a persistent B+-tree with low tail latency. *VLDB Endowment* 13, 12 (2020), 2634–2648.

[17] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[18] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard P. Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, CA, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*. ACM, Indiana, 143–154. https://doi.org/10.1145/1807128.1807152

[20] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. 2021. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, 503–516. https://doi.org/10.1145/3445814.3446744

[21] Jens Dittrich, Joris Nix, and Christian Schön. 2021. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. *Proc. VLDB Endow.* 15, 3 (2021), 527–540. https://doi.org/10.14778/3494124.3494136

[22] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Canada, 478–493. https://doi.org/10.1145/3341301.3359637

[23] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward R. Zayas. 2008. FlexVol: Flexible, Efficient File Volume Virtualization in WAFL. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, Boston, 129–142. http://www.usenix.org/events/usenix08/tech/fullpapers/edwards/edwards.pdf

[24] Goetz Graefe. 2010. A Survey of B-tree Locking Techniques. *ACM Trans. Database Syst.* 35, 3 (2010), 16:1–16:26.

[25] Goldstein, Jonathan and Abdelhamid, Ahmed and Barnett, Mike and Burckhardt, Sebastian and Chandramouli, Badrish and Gehring, Darren and Lebeck, Niel and Meiklejohn, Christopher and Minhas, Umar Farooq and Newton, Ryan and others. 2020. Ambrosia: Providing Performant Virtual Resiliency for Distributed Applications. *Proceedings of the VLDB Endowment* 13, 5 (2020), 588–601.

[26] Graefe, Goetz. 2012. A survey of B-tree Logging and Recovery Techniques. *ACM Transactions on Database Systems (TODS)* 37, 1 (2012), 1–35.

[27] Graefe, Goetz and others. 2011. Modern B-tree techniques. *Foundations and Trends® in Databases* 3, 4 (2011), 203–402.

[28] Greg Roelofs, Mark Adler. 2022. Zlib compression library. https://zlib.net/.

[29] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2011. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX, CA, 91–103. http://www.usenix.org/events/fast11/tech/techAbstracts.html#Gupta

[30] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. 2020. WAL-SSD: Address Remapping-Based Write-Ahead-Logging Solid-State Disks. *IEEE Trans. Computers* 69, 2 (2020), 260–273.

[31] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *International Conference on Management of Data (SIGMOD)*. ACM, Portland, 877–892. https://doi.org/10.1145/3318464.3389716

[32] Pedro Holanda, Stefan Manegold, Hannes Mühleisen, and Mark Raasveldt. 2019. Progressive Indexes: Indexing for Interactive Data Analysis. *Proc. VLDB Endow.* 12, 13 (2019), 2366–2378. https://doi.org/10.14778/3358701.3358705

[33] Kaisong Huang, Yuliang He, and Tianzheng Wang. 2022. The Past, Present and Future of Indexing on Persistent Memory. *Proc. VLDB Endow.* 15, 12 (2022), 3774–3777. https://www.vldb.org/pvldb/vol15/p3774-wang.pdf

[34] Shehbaz Jaffer, Stathis Maneas, Andy A. Hwang, and Bianca Schroeder. 2019. Evaluating File System Reliability on Solid State Drives. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, WA, 783–798. https://www.usenix.org/conference/atc19/presentation/jaffer

[35] Theodore Johnson and Dennis E. Shasha. 1989. Utilization of B-trees with Inserts, Deletes and Modifies. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, USA, 235–246. https://doi.org/10.1145/73721.73745

[36] Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. 2019. A specialized B-tree for concurrent datalog evaluation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, WA, 327–339. https://doi.org/10.1145/3293883.3295719

[37] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, USA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[38] Linux Kernel. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[39] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, CA, 161–177. https://www.usenix.org/conference/osdi22/presentation/kim

[40] KIOXIA CD8-V Specification. 2023. KIOXIA. https://www.kioxia.com.cn/zh-cn/business/ssd/data-center-ssd/cd8-v.html.

[41] Andreas Kipf, Damian Chromejko, Alexander Hall, Peter A. Boncz, and David G. Andersen. 2020. Cuckoo Index: A Lightweight Secondary Index Structure. *Proc. VLDB Endow.* 13, 13 (2020), 3559–3572. https://doi.org/10.14778/3424573.3424577

[42] Bradley C. Kuszmaul, Matteo Frigo, Justin Mazzola Paluska, and Alexander (Sasha) Sandler. 2019. Everyone Loves File: File Storage Service (FSS) in Oracle Cloud Infrastructure. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, WA, 15–32. https://www.usenix.org/conference/atc19/presentation/kuszmaul

[43] Lee, Leon and Xie, Siphrey and Ma, Yunus and Chen, Shimin. 2022. Index checkpoints for instant recovery in in-memory database systems. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1671–1683.

[44] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *International Conference on Data Engineering (ICDE)*. IEEE Computer Society, France, 185–196. https://doi.org/10.1109/ICDE.2018.00026

[45] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Canada, 447–461. https://doi.org/10.1145/3341301.3359628

[46] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Australia, 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[47] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. LLAMA: A Cache/Storage Subsystem for Modern Hardware. *Proc. VLDB Endow.* 6, 10 (2013), 877–888. https://doi.org/10.14778/2536206.2536215

[48] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, USA, 673–687. https://www.usenix.org/conference/atc21/presentation/li-yongkun

[49] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090. https://doi.org/10.14778/3384345.3384355

[50] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+tree with HTM. In *International Conference on Parallel Processing (ICPP)*. ACM, Japan, 101:1–101:10. https://doi.org/10.1145/3337821.3337827

[51] Arlino Magalhães, Angelo Brayner, José Maria Monteiro, and Gustavo Moraes. 2021. Indexed Log File: Towards Main Memory Database Instant Recovery. In *International Conference on Extending Database Technology (EDBT)*. OpenProceedings.org, Cyprus, 355–360. https://doi.org/10.5441/002/edbt.2021.34

[52] Magalhaes, Arlino and Monteiro, Jose Maria and Brayner, Angelo. 2021. Main Memory Database Recovery: A survey. *ACM Computing Surveys (CSUR)* 54, 2 (2021), 1–36.

[53] Darko Makreshanski, Justin J. Levandoski, and Ryan Stutsman. 2015. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. *Proc. VLDB Endow.* 8, 11 (2015), 1298–1309. https://doi.org/10.14778/2809974.2809990

[54] MariaDB Foundation. 2023. MariaDB. https://mariadb.org/.

[55] Melnik, Sergey and Gubarev, Andrey and Long, Jing Jing and Romer, Geoffrey and Shivakumar, Shiva and Tolton, Matt and Vassilakis, Theo. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 330–339.

[56] Micron 7450Pro Specification. 2023. Micron. https://www.micron.com/products/ssd/product-lines/7450.

[57] Microsoft. 2023. SQL Server. https://www.microsoft.com/en-us/sql-server/.

[58] Mohan, Chandrasekaran and Haderle, Don and Lindsay, Bruce and Pirahesh, Hamid and Schwarz, Peter. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks using Write-ahead Logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[59] MongoDB, Inc. 2023. WiredTiger. https://www.mongodb.com/docs/manual/core/wiredtiger/.

[60] NetApp. 2023. NetApp Thin Provisioning Concept. https://docs.netapp.com/us-en/ontap/concepts/thin-provisioning-concept.html.

[61] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research (CIDR)*. www.cidrdb.org, The Netherlands. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[62] Oracle. 2023. MySQL. https://www.mysql.com/.

[63] Oracle. 2023. MySQL Checkpoints. https://dev.mysql.com/doc/refman/5.7/en/innodb-checkpoints.html.

[64] Oracle. 2023. MySQL Compression. https://dev.mysql.com/doc/refman/8.0/en/innodb-compression-internals.html.

[65] Oracle. 2023. Oracle database. https://www.oracle.com/.

[66] O'Neil, Patrick and Cheng, Edward and Gawlick, Dieter and O'Neil, Elizabeth. 1996. The Log-structured Merge-Tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[67] Kai Qian, Letian Yi, and Jiwu Shu. 2011. ThinStore: Out-of-Band Virtualization with Thin Provisioning. In *International Conference on Networking, Architecture, and Storage (NAS)*. IEEE Computer Society, China, 1–10. https://doi.org/10.1109/NAS.2011.39

[68] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, USA, 69–82. https://www.usenix.org/conference/fast22/presentation/qiao

[69] Rodeh, Ohad and Bacik, Josef and Mason, Chris. 2013. BTRFS: The Linux B-tree Filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.

[70] Samsung PM9A3 Whitepaper. 2023. Samsung. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[71] Sauer, Caetano and Graefe, Goetz and Härder, Theo. 2018. FineLine: Log-structured Transactional Storage and Recovery. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2249–2262.

[72] Scaleflux. 2023. Computational storage drive with built-in transparent compression. https://scaleflux.com/.

[73] Scaleflux. 2023. ScaleFlux 3000-series SSDs. https://scaleflux.com/products/csd-3000/.

[74] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, USA, 219–238. https://www.usenix.org/conference/osdi21/presentation/shen

[75] SQLite. 2023. SQLite. https://www.sqlite.org/.

[76] The PostgreSQL Global Development Group. 2023. PostgreSQL. https://www.postgresql.org/.

[77] Vohra, Deepak and Vohra, Deepak. 2016. Apache Parquet. *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools* (2016), 325–335.

[78] 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 *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, USA, 429–444. https://www.usenix.org/conference/fast22/presentation/wang

[79] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick P. C. Lee. 2020. Austere Flash Caching with Deduplication and Compression. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, USA, 713–726. https://www.usenix.org/conference/atc20/presentation/wang-qiuping

[80] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *International Conference on Management of Data (SIGMOD)*. ACM, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[81] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *International Conference on Management of Data (SIGMOD)*. ACM, USA, 473–488. https://doi.org/10.1145/3183713.3196895

[82] William E Wright. 1985. Some average performance measures for the B-tree. *Acta Informatica* 21 (1985), 541–557.

[83] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. 2010. Efficient B-Tree Based Indexing for Cloud Data Processing. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 1207–1218. https://doi.org/10.14778/1920841.1920991

[84] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *USENIX Annual Technical Conference (ATC)*. USENIX Association, USA, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[85] Yu, Geoffrey X and Markakis, Markos and Kipf, Andreas and Larson, Per-Åke and Minhas, Umar Farooq and Kraska, Tim. 2022. TreeLine: an Update-in-place Key-value Store for Modern Storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[86] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *International Conference on Management of Data, Philadelphia (SIDMOD)*. ACM, USA, 1655–1669. https://doi.org/10.1145/3514221.3526130

[87] You Zhou, Qiulin Wu, Fei Wu, Hong Jiang, Jian Zhou, and Changsheng Xie. 2021. Remap-SSD: Safely and Efficiently Exploiting SSD Address Remapping to Eliminate Duplicate Writes. In *USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, USA, 187–202. https://www.usenix.org/conference/fast21/presentation/zhou