Towards Encrypted Data Compression with Computational Storage Drives

Linsen Ma
Rensselaer Polytechnic Institute
Troy, USA
malinsen19980103@gmail.com

Rui Xie
Rensselaer Polytechnic Institute
Troy, USA
xier2@rpi.edu

Feng Chen
Indiana University Bloomington
Bloomington, USA
fchen25@iu.edu

Xiaodong Zhang
The Ohio State University
Columbus, USA
zhang@cse.ohio-state.edu

Tong Zhang
Rensselaer Polytechnic Institute
Troy, USA
tzhang@ecse.rpi.edu

Abstract

Modern data center systems need to achieve several critical goals in security, performance, and cost efficiency. However, realizing these goals simultaneously is highly challenging. In secure data storage systems, a common practice is to first compress and encrypt data on the host side and then transmit it to the storage system using a log-based structure. This approach, unfortunately, leads to increased complexity and performance penalty. As an emerging storage technology, Computational Storage Drives (CSD) can not only offload heavy computation burdens to storage device hardware, but also provide a virtualized logical storage space, creating new optimization opportunities. In this paper, we showcase two unique opportunities enabled by the new CSD technology in data storage management. By replacing ordinary SSDs with CSDs, we can realize efficient one-to-one mapping from host-side blocks to storage-side CSD blocks, eliminating the need for a complex log-based structure and the associated heavy-cost operations, such as garbage collections (GC). Moreover, with a carefully redesigned data format in each compression unit, CSDs can transparently remove redundant data across encrypted snapshots in data-intensive environments, such as databases. We have developed a prototype and conducted experiments on ScaleFlux's CSD 3000 devices to demonstrate the efficacy of these solutions. We hope that our system investigations in this work provide valuable insight into CSDs and inspire researchers and practitioners to explore additional cases for adopting CSDs to improve the performance and productivity of data center systems.

CCS Concepts

• **Information systems → Storage architectures**; • **Software and its engineering → Software system structures**.

Keywords

Computational storage drive, Data compression, Encryption, Flash memory, Solid state drives.

SSDBM 2026, San Diego, CA, USA

ACM ISBN 979-8-4007-2708-5/26/08
https://doi.org/10.1145/3828820.3828826

ACM Reference Format:
Linsen Ma, Rui Xie, Feng Chen, Xiaodong Zhang, and Tong Zhang. 2026. Towards Encrypted Data Compression with Computational Storage Drives. In *The International Conference on Scalable Scientific Data Management 2026 (SSDBM 2026), August 11–13, 2026, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3828820.3828826

1 Introduction

Modern data center systems are required to achieve several critical goals: security, performance, and cost efficiency. Although each of these goals is equally important, realizing one often comes at the cost of another, which poses significant challenges for us to create a well-balanced system in data center environments.

In data storage systems, this issue is particularly prominent. *Encryption* and *compression* are two major tools for securing data and reducing cost, however, it is non-trivial to integrate them efficiently in one system, because encryption randomizes data and destroys data compressibility, leaving little room for compression to reduce data storage cost. A commonly adopted solution is to first compress data and then encrypt it before transmitting it to the storage system. However, supporting such a data flow requires a complex log-based structure and Garbage Collection (GC) for data management, which unfortunately introduces significant I/O amplification and performance penalty [22, 24, 36, 41]. As a recent advancement in data storage technologies, *Computational Storage Drive* (CSD) brings numerous new opportunities, which could address these challenges and help us achieve the three important goals all at once.

The fundamental idea behind CSD, empowering data storage devices with computing capability, dates back more than 20 years [4, 29, 38]. An important breakthrough in recent years is the maturity of modern CSDs with *transparent compression* [25, 44, 45]. This new category of storage devices, represented by products from ScaleFlux [45] and DapuStor [25], integrate a powerful hardware engine on device to compress data on the I/O path, without introducing noticeable overhead on requests. By hiding heavy-cost compression behind the standard Logical Block Address (LBA) interface, it transparently compresses data in units of the standard 4 KB blocks and presents an expanded "logical" storage space to the host system.

However, in the scenario of encrypted data storage, a long-standing conception is that such hardware-level transparent compression technologies are "incompatible" with encryption, because

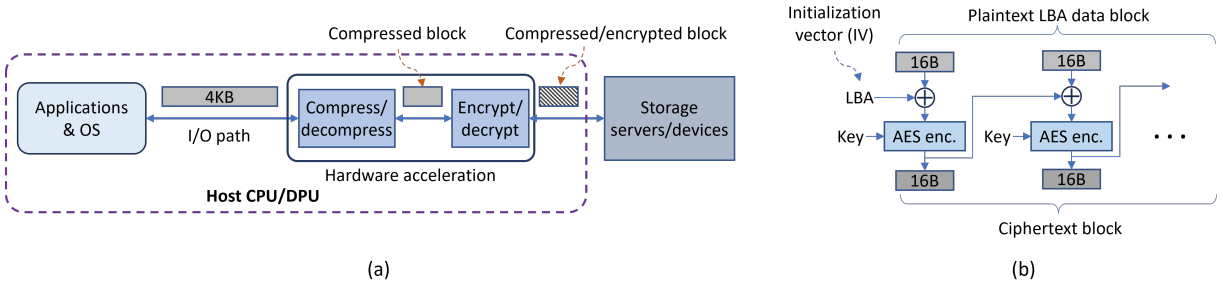

Figure 1: An illustration of (a) the I/O path of encrypted data compression and (b) AES-XTS encryption with LBA as initiation vector (IV).

encrypted data are simply random bits and cannot be further compressed, which renders these CSDs with transparent compression merely useful in such environments. In this work, we will show that this is *not* true. In fact, contrary to these previously widely accepted misconceptions, our studies indicate that *a proper use of CSD with optimizations in systems and applications can effectively help us achieve the three above-mentioned goals in a highly efficient manner*.

In this paper, we particularly exploit two new opportunities enabled by CSD with transparent compression, in the scenarios of data management and databases. We hope that the two cases presented in this paper can provide valuable insight and inspire researchers and practitioners in the community to develop innovative use of CSDs in data center systems.

We make the following contributions in this paper: (1) we investigate the unique properties of modern CSDs and their unconventional role in encrypted data compression, (2) we explore the potential benefits and different means of utilizing high-speed transparent compression on device, and (3) we develop two case studies to demonstrate the unique opportunities of leveraging the virtual storage space enabled by CSDs in data management and snapshot handling.

The remainder of this paper is organized as follows. Section 2 provides background. Section 3 and Section 4 present the two opportunities enabled by CSD. Section 5 details the system implementation and evaluation results. Section 6 discusses the related work. The final section concludes this paper.

2 Background

2.1 Encrypted Data Compression

Data compression and encryption have long been considered compute-intensive operations. Recent hardware advancements have made tremendous progress in accelerating these operations. Modern CPUs, such as Intel's Sapphire Rapids and IBM's Power9/z15 [2], integrate on-chip high-speed compression accelerators; DPUs, such as Amazon's Nitro [8] and NVIDIA's Bluefield [13], also have embedded compression/encryption accelerators for data center systems. These emerging technologies allow computing infrastructures to seamlessly integrate a hardware-accelerated compression-encryption pipeline into the existing I/O stack, as illustrated in Figure 1(a). It enables *encrypted data compression* for IOPS-critical

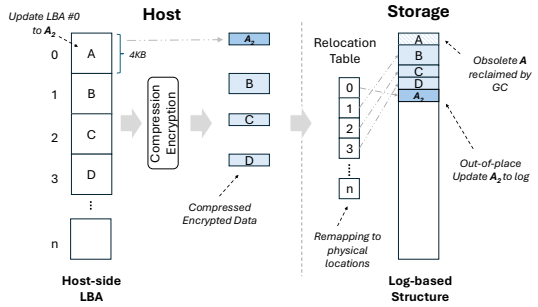

Figure 2: An illustration of log-based management of variable-length compressed/encrypted blocks.

block-level storage services, such as Amazon Instance Store and Elastic Block Store [9], which are in strong demand among their cloud-computing customers.

AES-XTS [18, 40] is widely used in practice for data storage encryption. It combines AES (advanced encryption standard) [3] with the principle of tweakable encryption [32]. As shown in Figure 1(b), AES-XTS uses the LBA of each data block as the initialization vector (IV) input to the AES encryption process. As writing the same data content to different LBAs would produce different encryption output, it significantly enhances data security. However, from the perspective of data compression, AES-XTS makes it very difficult, if not impossible, to implement data deduplication [50] to reduce storage cost through data compression, since each block now becomes unique despite the same data content.

2.2 Block Management

Given the unpredictable and dynamically varying data compressibility, the storage management of compressed/encrypted data blocks essentially behaves similarly to a key-value store: Each data item is a key-value pair, of which the *key* is the LBA of the block and the *value* is the variable-length compressed/encrypted data. Snapshots can be supported by attaching a version number to the key.

To achieve storage space efficiency, these variable-length key-value pairs need to be compactly stored on the storage media. Typically, a log-structured design [36, 41, 47] is used by appending securely compressed data blocks one after another on storage. Such an organization, though space-efficient, inherits many problems of

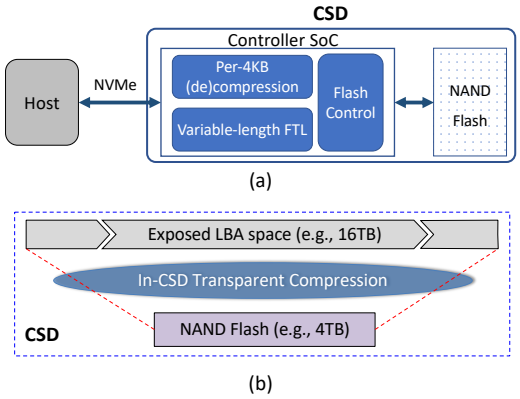

Figure 3: An illustration of (a) CSD with built-in transparent compression and (b) the expanded LBA space.

the log structure, such as time-consuming garbage collection (GC) operations and I/O amplifications, as illustrated in Figure 2.

Note that the whole process is completely transparent to host-side user applications. The storage system software is responsible for managing the placement and storage of compressed/encrypted data blocks, as well as other services, such as snapshots. From the host application's perspective, it only sees a standard 4 KB LBA interface, which hides the above-said complexities behind the scene. However, the performance impact induced by the log structure is a significant challenge in handling workloads with a large amount of small compressed data blocks.

2.3 CSD and Transparent Compression

Computational Storage Drives (CSDs) have emerged as a significant breakthrough in data storage technologies in recent years. CSD embeds computational capabilities directly in storage devices. A distinguishing feature of modern CSDs is *transparent compression*, which is performed entirely at the device level without explicit involvement from host-side software. Commercial offerings, such as ScaleFlux's CSD [45] and DapuStor's drives [25], incorporate specialized hardware accelerators in their controller chips to compress and decompress data seamlessly on the storage I/O path.

As illustrated in Figure 3(a), a typical CSD integrates a dedicated hardware accelerator within the storage controller System on Chip (SoC), which automatically compresses and decompresses data blocks (typically 4 KB in size) when being written or read. This hardware-accelerated process adds minimal latency, only a few microseconds, which is negligible compared to typical NAND flash memory latencies (50 μs or more for reads, and around 1 ms for writes). Thus, modern CSDs are capable of delivering nearly identical performance in terms of IOPS and latency as conventional SSDs, with substantially more usable storage capacity and cost savings.

To efficiently utilize compression, modern CSDs provide an expanded LBA space, as illustrated in Figure 3(b). The expanded LBA space, characterized by an expansion factor α_{exp} (e.g., 2× or 4×), allows CSDs to accommodate compressed data flexibly. For example, with an LBA space expansion factor $\alpha_{exp} = 4$, a CSD with 1 TB physical flash memory capacity exposes a 4 TB logical storage space. However, a higher expansion factor increases the complexity and the mapping table size of the Flash Translation Layer (FTL) on device, demanding additional on-device DRAM size. Due to practical constraints, such as device form factor and production cost, expansion factors are usually limited. Typical commercial CSDs set an upper limit of α_{exp} at 2 or 4.

A unique benefit of CSD with transparent compression is that it provides a "virtualized" logical storage space. This enables us to aggressively *reserve* certain storage space in advance and create a *sparse* data structure, without worrying about wasting physical storage capacity. For example, a logical 4 KB block can be intentionally filled up by padding 3 KB of zeroes in the block, while only consuming 1 KB of storage space physically. This property is the technical foundation that enables the two novel optimization opportunities as we show later in this paper.

3 Opportunity I: CSD in Data Management

3.1 Motivation

Traditionally, encrypted data compression involves host-side compression followed by encryption and then transmission to the storage side. Such an approach mitigates possible data breach during transmission and facilitates end-to-end data security, but it results in *variable-length* compressed data chunks, which cannot be perfectly aligned in storage.

As previously illustrated in Figure 2, a 4 KB logical block on the host side is compressed into a smaller size, then encrypted, and finally transmitted to the storage. We call them *compressed/encrypted blocks*. Since the data chunks received at the storage side are of variable sizes, for the purpose of saving storage capacity, these compressed/encrypted blocks need to be packed tightly one after another in a log-based structure [36, 41], which is space-efficient but unfortunately brings two performance issues:

First, since the variable-length data chunks cannot be perfectly aligned due to the variable sizes, it demands an extra mapping structure to map each host-side 4 KB logical block to the physical location where it is stored. Second, updating a logical data block becomes more costly. Because the data cannot be updated in place, the new data needs to be appended in the log, while marking the original data as "obsolete"; A Garbage Collection (GC) procedure runs periodically to reclaim the space occupied by the obsolete data in an asynchronous way. Note that the GC here refers to the host-side recycling of logical blocks, rather than the device-level GC for flash memory [5, 16]. As a result, these additional operations cause amplified storage I/Os, increased overhead, more complexity, and reduced performance.

The virtualized storage provided by CSDs enables a unique opportunity to simplify storage management: As CSDs can easily remove *zero* bits at the device level, a data block compressed and encrypted on the host side can be artificially bloated with zeroes and stored in a CSD block, occupying the entire 4 KB logical CSD block but only consuming a fraction of the physical storage space on device.

As an example in Figure 4, given a 4 KB block on the host side, say A, it is first compressed and encrypted, resulting in a 1 KB data chunk, A', which is later transmitted to the CSD; On the storage

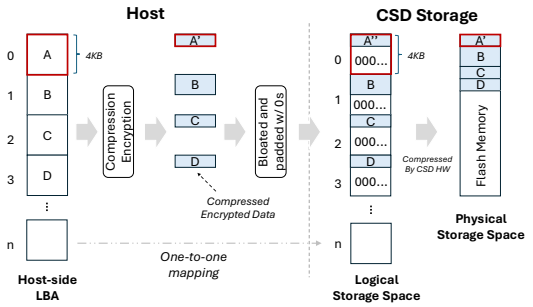

Figure 4: An illustration of the management of bloated compressed/encrypted blocks with CSD.

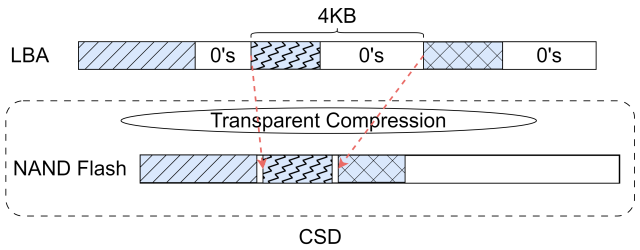

Figure 5: An illustration of sparsely filled 4 KB LBA block on CSD with built-in transparent compression.

side, A' is first padded with 3 KB of zeros, becoming an artificially bloated 4 KB block, A'', which is then stored in the CSD. Thanks to the transparent compression capability, A'' occupies a full 4 KB *logical* CSD block but only consumes 1 KB of *physical* storage in flash memory.

Note that the purpose of adopting CSD here is not to provide additional space saving, but to enable a *one-to-one direct mapping* from the host-side block to the CSD block ($A \rightarrow A''$ in the previous example). As a result, each host-side data block can be directly *in-place updated* on the CSD storage space, which fundamentally eliminates the need for the log structure and GC, as well as the incurred I/O amplification and other management overhead on the host side.

It is worth pointing out that this approach is enabled by and only feasible on CSDs with transparent compression, because CSDs can automatically compress away the padded zeros (see Figure 5). With conventional SSDs, in contrast, storing such a zero-padded 4 KB block still consumes 4 KB physical storage capacity, regardless of data content (even all zeros). Thanks to the high-speed PCIe interface, transferring the zero padding introduces only minimal additional latency compared with the overall I/O path.

3.2 CSD-assisted Data Management

The above-said approach directly allows us to map a user-viewable 4 KB logical LBA block on the host side, in a zero-padded, compressed and encrypted format, to a 4 KB CSD block on the storage side. However, such a one-to-one mapping is implicitly limited by the *expansion constraint* of the CSD:

Due to the limited RAM capacity on drive, the CSD device cannot maintain an excessively large mapping table on device. Thus, the logical storage space of a CSD device typically uses a conservative expansion constraint, such as 2 or 4, meaning that a CSD with 1 TB physical storage capacity can expose a 2 TB or 4 TB logical storage (LBA) space for use by the host. However, in reality, the runtime data compression ratio could be much higher, such as 10:1, meaning a substantial waste of physical storage capacity if we adopt a one-to-one mapping. Taking a CSD with 1 TB physical storage and an expansion constraint of 2 as an example, due to the one-to-one mapping requirement as described above, the CSD can only store 2 TB of user data, using only 0.2 TB of physical flash storage space while leaving the remaining storage space unused.

Performance vs. capacity. Let β denote the average data compression ratio. If we strictly use a one-to-one mapping as described above, to fully utilize the internal physical flash storage capacity, the CSD's storage space expansion factor α_{exp} must be greater than β, which however cannot always be possible in reality. To address this issue, we need to make a necessary compromise in the design: some compressed blocks need to share one logical CSD block.

The challenge is that sharing a logical CSD block could cause performance problems: With multiple compressed/encrypted blocks in a logical CSD block, I/O amplification would happen upon reads and writes, because only a portion of the block is needed. On the other hand, having a compressed block occupy an entire logical CSD block would demand more logical block space, leaving physical storage space underutilized, which is also undesirable.

Differentiated Data Management. Our solution is to differentiate blocks according to their access locality and handle them differently. For example, only allowing hot data blocks that need to be frequently accessed to exclusively occupy a logical CSD block, while packing multiple cold data blocks in one logical CSD block to better utilize the space. With such a hybrid approach, we can achieve the purpose of accommodating a large β with a smaller α.

To accommodate a wide range of compression ratios under the CSD's storage space expansion constraint, we propose to partition the CSD storage space into three logical regions:

- *Hot region*: Each 4 KB logical CSD block hosts only one 4 KB host block, which is compressed, encrypted, and padded with zeros. Blocks in this region can be directly *in-place updated* without causing any GC operations. This region stores update-intensive data, i.e., "hot" blocks.
- *Warm region*: For each 4 KB logical CSD block, we define its *fill factor* ζ_{fill} as the percentage of its 4 KB space being occupied by real data. In other words, $1 - \zeta_{fill}$ of the 4 KB space is filled with all zeros. Given the CSD's space expansion factor α_{exp}, each logical CSD block in this region is filled with one or multiple compressed/encrypted blocks, subject to the fill factor of $\zeta_{fill} = 1/\alpha_{exp}$. For example, assume a CSD could expand its logical storage space with $\alpha_{exp} = 2$, the fill factor ζ_{fill} is set to 50%. This region stores securely compressed blocks with moderate amount of updates, i.e., "warm" blocks.
- *Cold region*: Each 4 KB logical CSD block packs as many compressed blocks as possible, without any fill factor constraint.

This region stores compressed/encrypted blocks with low update probabilities, i.e., "cold" blocks.

The hotness of data blocks is estimated based on their update intensity. Initially, all blocks are considered "warm" and are stored in the warm region. During runtime, data with update intensity greater than the update intensity threshold $b_{h/w}$ is promoted to the hot region; Data blocks with update intensity lower than update intensity threshold $b_{w/c}$ are downgraded to the cold region. As mentioned above, logical CSD blocks in the warm or cold regions contain more than one compressed/encrypted block, and GC is initiated when a logical CSD block contains over 70% invalid obsolete data.

Setting region size. A challenge is how to properly set the sizes of the three regions. Let C_{hot}, C_{warm}, and C_{cold} denote the capacity of the hot, warm, and cold regions, respectively. In adaptation to the runtime data compressibility and data access characteristics, we need to dynamically adjust the ratio of $C_{hot} : C_{warm} : C_{cold}$ to keep as many update-intensive data blocks in the hot region as possible, while complying with the CSD's space expansion factor constraint. To this end, we propose a dynamic hotness-based region management algorithm, following the principle of linear regression.

As described in Algorithm 1, after a certain number (10 million by default) of update operations, we recompute the current target ratios h_t, w_t, and c_t in *get_target_ratio*, and right-shift the update intensity thresholds $b_{h/w}$ and $b_{w/c}$, and the update intensity counter b_i, to avoid counter overflow and to phase out the impact of older accesses. During this windowed period, every 1 million update operations being processed, the system dynamically adjusts the values of $b_{h/w}$ and $b_{w/c}$. Using the principle of linear regression, we implement this fine-tuned dynamic adjustment mechanism, ensuring a gradual alignment with the target ratio assignments.

Algorithm 1 Dynamic hotness-based region management

h_t, w_t, c_t	:	Target region capacity ratios
h_r, w_r, c_r	:	Current capacity ratios
U_{ops}	:	Update operation counter
$b_{h/w}$, $b_{w/c}$	:	Update intensity thresholds
$l_{h/w}$, $l_{w/c}$	:	Learning rate for $b_{h/w}$ and $b_{w/c}$
b_i	:	Update intensity of each block
μ_u, δ_u	:	Mean and deviation of all blocks' update intensity
β	:	Average data compression ratio
α_{exp}	:	CSD's storage space expansion factor

```
1: function Update_Thresholds(U_ops)
2: if U_ops mod 1M == 0 then
3:    if U_ops mod 10M == 0 then
4:       h_t, w_t, c_t = get_target_ratio(μ_u, δ_u, β, α_exp)
5:       b_{h/w} = b_{h/w}/2, b_{c/w} = b_{c/w}/2, b_i = b_i/2
6:    else
7:       b_{h/w} = b_{h/w} + l_{h/w} × (h_r − h_t)
8:       b_{w/c} = b_{w/c} + l_{w/c} × (w_r − w_t)
9:    end if
10: end if
11: end function
```

The function *get_target_ratio* dynamically determines the capacity distribution among the hot, warm, and cold storage regions. To simplify the model and improve stability, we fix the cold region's capacity at 20% of the total storage. Specifically, data blocks are classified as *cold* if their update intensity b_i falls below the threshold $\mu_u - k \cdot \delta_u$, where μ_u is the mean update intensity across all blocks and δ_u is the corresponding standard deviation. The threshold factor k is empirically selected to ensure approximately 20% of the blocks qualify as cold, thus aligning the cold region to occupy 20% of total capacity. Accordingly, the function can obtain the value of target c_t, based on which the function can calculate the value of h_t and w_t as:

$$h_t = \frac{(\alpha_{exp} - 1) \cdot \beta}{\beta - \alpha_{exp}} c_t,$$
$$w_t = 1 - h_t - c_t.$$

The rationale behind this process is to estimate a proper allocation that can fit the data that are classified in three categories in the given CSD storage space according to the expansion constraint and compression ratio.

Data relocation. When data residing in a logical CSD block encounters a dynamic region resizing, it remains in the logical CSD block until it is updated. Upon being updated, the data is assigned to a logical CSD block in the new region, and the old data is marked as invalid. Once a logical CSD block accumulates 70% invalidated compressed/encrypted blocks, GC is triggered. During GC, the GC thread moves the remaining valid data in the block to new logical CSD blocks that belong to the new hotness level. Again, note that the GC here refers to the garbage collection on the host-side recycling of logical CSD blocks, rather than the device-level GC for flash memory.

4 Opportunity II: CSD in Low-cost Snapshot

4.1 Motivation

In data centers, snapshots [31] are important for data backup, protection, and disaster recovery. However, within the scenario of encrypted data compression, efficient snapshot storage becomes challenging.

As mentioned earlier, AES-XTS [18, 40] combines AES (advanced encryption standard) [3] with the principle of tweakable encryption [32]. As AES-XTS uses Logical Block Address (LBA) as an initialization vector, snapshots of the same block that are stored in different locations could be drastically different after encryption, although the actual difference between pre-encryption snapshots could be minimal. Consequently, traditional methods for deduplication and cross-snapshot compression become largely ineffective with AES-XTS. Although snapshot systems could use techniques, such as copy-on-write [35], to improve space efficiency, full-storage snapshots are still required periodically. In this case study, we focus on discussing the application of CSDs in the scenarios of full-storage snapshots rather than incremental updates.

The transparent compression feature of CSDs offers a new opportunity to overcome this problem. We find that despite encryption-induced randomization, *substantial internal redundancy still remains across encrypted snapshot versions*. Thus, if multiple snapshot versions of the same data block were purposefully placed within the same CSD compression unit, the CSD's device-level transparent compression still could effectively remove such data redundancies

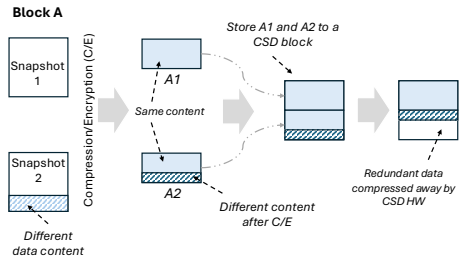

Figure 6: An illustration of cross-snapshot content redundancy and data compression with CSD.

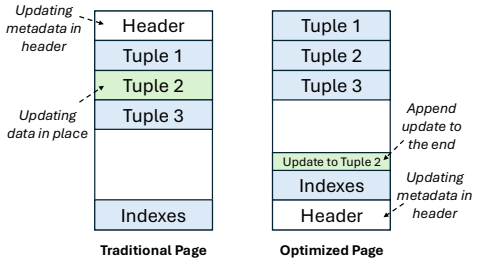

Figure 7: Comparison between conventional and modified B+ tree page layout.

automatically, significantly reducing storage demands with snapshots.

4.2 CSD-assisted Low-cost Snapshot

Based on the finding that even after compression and encryption, different snapshot versions of the same host block could still have a high degree of similarity, we can leverage CSDs to reduce snapshot-induced storage cost overhead.

As illustrated in Figure 6, for a LBA block, the difference between its 1st and 2nd snapshots is only a few bytes in the tail. With the same compression/encryption algorithm and the same encryption initialization vector (i.e., LBA), the two compressed/encrypted snapshots share an identical segment at the front. Leveraging the transparent compression capability of CSD, we can reduce the storage cost by storing multiple snapshot versions on the per-LBA basis to the CSD.

Critical issue. Since the length of the identical segment depends on the location of updates within the block, how the data is updated would significantly influence the achievable compression ratio between two encrypted snapshots of a data block. Thus, the compressible cross-snapshot redundancy depends on how applications change the LBA block from one snapshot to another. For example, if applications modify the very first or the last byte of the block, the cross-snapshot redundancy would be none or almost 100%. Therefore, it becomes important to optimize the application's *data placement strategy* to purposefully create opportunities for CSDs to remove cross-snapshot redundancy.

A case in databases. We demonstrate this potential opportunity by using database's index files as an example. Relational databases typically use B+ tree index for data storage management, where the B+ tree page size is a multiple of 4 KB (e.g., 16 KB in MySQL/InnoDB, and 8 KB in PostgreSQL and Oracle). Conventional B+ tree page format results in low cross-snapshot redundancy, because it maintains critical metadata in the *header* of each page, which are frequently updated (see Figure 7). Moreover, randomly deleting and updating tuples that locate near the front of an LBA block would also significantly reduce the cross-snapshot redundancy.

To create opportunities for compression, we intend to keep the updates near the end of an LBA block. We propose an optimized B+ tree page data layout as illustrated in Figure 7: (1) We move the critical metadata to the end of each page, and (2) based on the fact that B+ tree pages are typically 30~50% empty [21], we use the intra-page empty space as a temporary buffer to log tuple updates, converting in-place updates to append-only updates. Only when a page becomes full, it is reformatted by merging the intra-page logged tuple updates into their home tuples. In this way, we can keep the majority portion of the page unchanged, purposefully creating opportunities for effective compression on CSD.

4.3 Discussion

Our snapshot optimization preserves data similarity across encrypted snapshot versions, which implies that coarse-grained, post-encryption, inter-version change patterns may be revealed. An adversary on the storage side who can observe the block traffic or have access to the raw ciphertext versions, device-internal information (e.g., compression ratio), or fine-grained timing behaviors, may analyze and infer whether a prefix of a block is changed or whether the updates are near the beginning or the end of a page. Although precisely determining the location of change in the original data format still remains difficult due to the unknown compression and encryption process on the host side, this mechanism does not provide update-locality confidentiality or snapshot-change privacy. The security goal of this snapshot optimization is limited to protecting user data, or data confidentiality. It is worth noting that this tradeoff is intentional and required to achieve storage efficiency. For environments where exposing certain level of update locality information is unacceptable, mitigation mechanisms, such as introducing randomness to disguise updates, could be used but they would reduce or eliminate the storage-saving benefit.

5 Implementation and Evaluation

5.1 Evaluation of CSD-based Data Management

We have implemented a prototype, called EZ-Store. As compressed data is of variable sizes, the storage management of compressed/encrypted data blocks essentially behave as a key-value store: Each compressed data block is a pair of *key*, which is the LBA of the data block, and *value*, which is the variable-length data chunk after being compressed and encrypted. Written in C++, EZ-Store uses a B^+ tree index and supports `GET`, `INSERT`, `UPDATE`, and `DELETE` operations. On the host side, the user data is compressed and encrypted using LZ4 and AES algorithms. EZ-Store organizes the compressed/encrypted data into hot, warm, and cold regions. Hot data are stored with one compressed/encrypted block per logical 4 KB CSD block to enable direct in-place updates, while warm and cold data may have multiple entries packed together within

Table 1: Compression statistics of different file types.

File type	Per-4KB compressed block size (Bytes)	
	Mean	Standard deviation
XLS	1,879	203
BMP	1,307	1,284

one logical block for high space utilization. The system tracks update intensity online and dynamically migrates data across regions. Garbage collection is triggered when the invalid-byte ratio of a logical block exceeds a configurable threshold, which is set to 50% by default in our prototype. The invalid-byte ratio is the total amount of obsolete compressed/encrypted entries to the logical block size. During garbage collection, EZ-Store relocates the remaining valid compressed/encrypted entries to new logical blocks and updates the corresponding B^+ tree metadata accordingly.

We compared EZ-Store with WiredTiger [49], RocksDB [39], and BlobDB [12]. Designed to handle big-data workloads, WiredTiger offers compatibility with various programming interfaces and good scalability. RocksDB is a high-performance, embeddable, and persistent key-value store that originated as a fork of the well-known LevelDB [19] from Google. BlobDB is an integral extension of RocksDB to efficiently manage large values or 'Blobs'. The evaluations were conducted on a server with a 2.30 GHz Intel Xeon Gold 5218 CPU with 16 cores/32 threads, 384 GB DDR4 DRAM, and a ScaleFlux CSD-3000 drive with built-in transparent compression.

5.1.1 Overall Performance. For the baseline performance evaluation and comparison, EZ-Store, RocksDB, and BlobDB used the same number (8 in this study) of GC/compaction threads, and we use the default setting in WiredTiger. We considered YCSB-like data access workloads with four different read/write ratios, including write-only, 80% writes, 50% writes, and read-only. We configured the size of each compressed/encrypted block according to the measurement from open corpus [27, 28, 46] as shown in Table 1 and chose XLS and BMP as two representatives.

Figure 8 shows the measured throughput and average latency. Under write-intensive workloads, EZ-Store significantly outperforms the others. For example, under 80% write and XLS-type data, EZ-Store achieves 167 Kops/s, representing 22%, 82%, and 67% increase compared with BlobDB, RocksDB, and WiredTiger, respectively. EZ-Store also exhibits correspondingly lower latencies than the others. As workloads become more read-intensive, the four data stores exhibit similar performance. This is because the read performance heavily depends on the SSD read latency/throughput and is weakly dependent on the data management/indexing structure.

5.1.2 Impact of Data Access Locality. We also test the throughput and write amplification under write-only workloads with different data access localities by using the widely adopted *Zipf* distribution. As the *Zipf* parameter α increases (from 1.2 to 2.2 in this study), the workload becomes more skewed with a higher degree of locality.

As shown in Figure 9, the throughput performance of all the four data stores improves as the data access locality increases, because a higher data access locality leads to a higher GC efficiency and hence lower write amplification. It also shows the evident write amplification reduction as data access locality increases. Compared with other alternatives, our approach consistently achieves the highest throughput, since its three-region data placement can effectively reduce the overall GC overhead and lower write amplification.

5.1.3 Impact of Compression Ratio. Figure 10 shows the throughput and write amplification of write-only workloads when the original 4 KB blocks are compressed/encrypted to 512 B, 1 KB, 1.5 KB, or 2 KB. We use synthesized data for the fixed-size compressed/encrypted data. The CSD logical storage space expansion factor α_{exp} is set to 2. The figure shows that the throughput performance improves as the size of compressed blocks increases. This is because as the size of compressed blocks increases, it results in less data fragmentation and hence lower GC frequency and write amplification. Compared with other alternatives, EZ-Store consistently achieves higher throughput and lower write amplification. Our approach also benefits the most from the increase of compressed block size. This is because as compressed block size approaches 2 KB, the value of β reduces to 2, and as a result, more 4 KB blocks on CSD fall into the hot region, leading to significantly lower GC frequency and write amplification.

5.1.4 Impact of Dataset Size. To assess the scalability with increasing dataset size, we conducted experiments with an average compressed/encrypted data block size set as 1 KB, varying the total number of data blocks from 10 M, 50 M, to 200 M. These configurations respectively correspond to total dataset sizes of 10 GB, 50 GB, and 200 GB. Results in Figure 11 show throughput degradation as the dataset size increases, but EZ-Store still outperforms the other data stores. As the dataset size increases, the update-induced stale data blocks would spread over a larger region, leading to a lower GC efficiency and hence a higher write amplification. As the dataset size increases, the write amplification of all the four data stores also increases, causing the degradation of data store throughput performance.

5.1.5 Impact of Space Amplification. In log-based data storage, the degree of write amplification depends on available storage space, causing a tradeoff between *space amplification* and *write amplification*. Space amplification is the ratio of the consumed CSD's LBA space to the live compressed/encrypted data size. At the cost of a higher space amplification, setting a higher space overprovisioning factor could improve the GC efficiency and hence reduce the write amplification. EZ-Store allows users to configure the permissible space amplification, and we accordingly studied its effect on write amplification and throughput performance. Since the CSD's space expansion factor α_{exp} is 2.0, we varied the EZ-Store permissible space amplification between 1.2 and 2.0.

Figure 12 shows the measured throughput and write amplification under different space amplification (SA), where the compressed/encrypted block size is 512 B, 1 KB, and 2 KB, respectively. When the compressed/encrypted block size is 512 B and 1 KB, EZ-Store's throughput slightly reduces as the space amplification decreases from 2 to 1.2. This decline is largely attributed to the increased frequency of background GC operations. Accordingly, the write amplification increases as the space amplification decreases, as shown in Figure 12. In contrast, when we increase the compressed/encrypted block size to 2 KB, EZ-Store treats each data block as "hot", because the CSD's storage space expansion factor

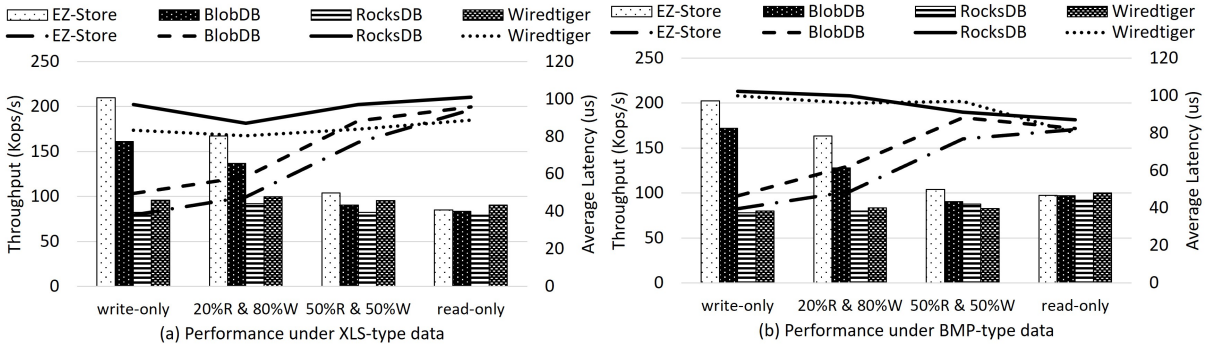

Figure 8: Overall performance of the data block management efficiency.

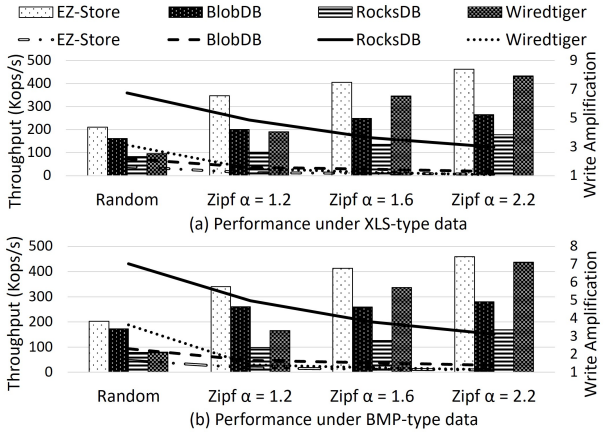

Figure 9: Impact of data access locality.

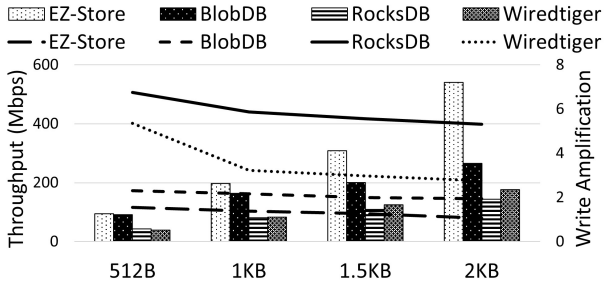

Figure 10: Impact of compressed block size.

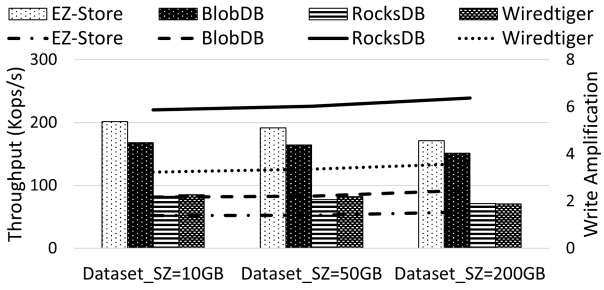

Figure 11: Impact of dataset size.

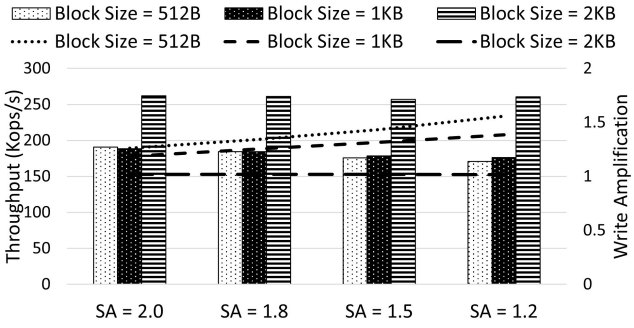

Figure 12: Impact of space amplification (SA).

is 2. As a result, every update request is served in-place, without triggering background GC operation. Therefore, the write amplification remains as 1, leading to a consistently high throughput as shown in Figure 12.

5.1.6 Impact of CSD Space Expansion Factor. Given the limited DRAM capacity inside storage drives due to the cost and form factor constraints, CSDs can only support a relatively small LBA space expansion factor (e.g., 2~4). In the experiments presented above, we set the CSD LBA space expansion factor α_{exp} as 2. For relatively small compressed/encrypted block sizes (e.g., 512 B and 1 KB), setting $\alpha_{exp} = 2$ results in a small hot region capacity, thereby increasing the degree of write amplification and GC operations in EZ-Store. We performed additional experiments to further evaluate the impact of CSD LBA space expansion factor.

Figure 13 shows the measured EZ-Store write amplification and throughput under the write-only workload, where the CSD's space expansion factor α_{exp} varies between 1.5 and 4. When compressed/encrypted block size is 512 B or 1 KB, the throughput increases as α_{exp} rises from 1.5 to 4.0. This is because a bigger CSD LBA space expansion factor enables a larger capacity of hot region, leading to a lower write amplification and hence higher write throughput performance. For instance, with an α_{exp} of 4.0 and compressed/encrypted block size of 1 KB, all the pages will be categorized as hot pages, eliminating the need for background GC operations. When compressed/encrypted block size is 2 KB, the

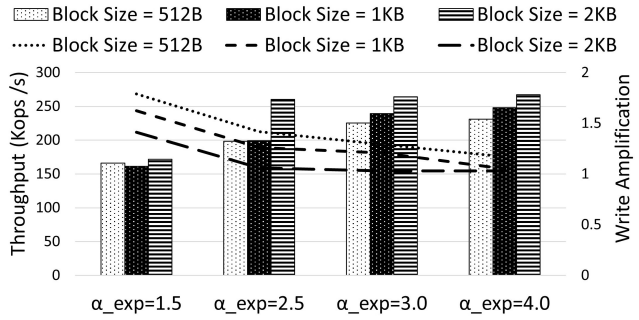

Figure 13: Impact of CSD LBA space expansion factor α_{exp}.

throughput improves when α_{exp} grows from 1.5 to 2.5, as shown in Figure 13. Since $\alpha_{exp} = 2$ is already large enough to eliminate the background GC operation when compressed/encrypted block size is 2 KB, further increasing α_{exp} cannot bring additional performance benefits.

5.2 Evaluation of CSD-based Snapshot Storage

In this section, we evaluate the proposed CSD-assisted snapshot management. We use the page format of MySQL/InnoDB as the baseline, and optimize its intra-page data placement using the scheme discussed in Section 4.2.

We implemented a simulator in C++, which constructs 16 KB MySQL/InnoDB-style pages with the specified fill factor. The simulator models the major components of an InnoDB page, including the page header, tuple storage area, free space, and page directory, and records the byte offset and size of each tuple in the page. In the baseline case, tuples are placed following the original InnoDB-style intra-page organization; In the optimized case, the same set of tuples is reorganized according to our CSD-aware page placement scheme, where in-place tuple updates are avoided and the page header is moved to the end of the page. The simulator randomly selects a fraction δ_p of tuples in each page to update between two consecutive snapshots, generates the corresponding before- and after-update page images, and measures the resulting cross-snapshot storage cost reduction at the 4 KB CSD block granularity.

The achievable snapshot storage cost reduction depends on two main factors, the cross-snapshot update intensity and page fill factor. We study their impact individually as follows.

5.2.1 Impact of Cross-snapshot Update Intensity. To evaluate the impact of cross-snapshot update intensity on the reduction of snapshot storage cost, we set the fill factor of each 16 KB MySQL/InnoDB page as 70%. Let δ_p denote the percentages of the tuples within each page that are randomly updated between two consecutive snapshots. We considered the scenarios with three different values of δ_p, from 5%, 10%, to 20%. As the value of δ_p increases, the content of two consecutive snapshots have more difference.

As shown in Figure 14, with the original MySQL/InnoDB page format, the cross-snapshot storage cost reduction quickly diminishes as δ_p increases. This is because as δ_p increases, each 4 KB LBA block is more likely to experience in-place updates closer to the front, resulting in less and less cross-snapshot content similarity after the host-side encrypted compression. In contrast, by

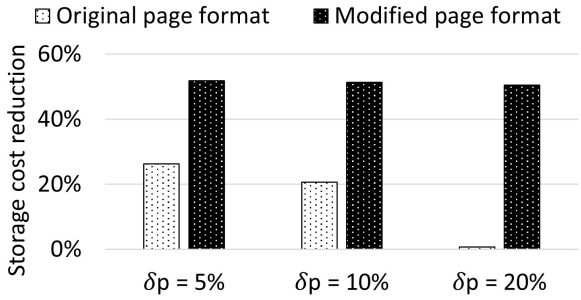

Figure 14: Snapshot storage cost reduction with the original and the modified page format with different update intensities.

avoiding in-place, intra-page updates and moving the page header to the end, as proposed in Section 4.2, our optimized page format can more effectively retain the similarity of cross-snapshot content after the host-side encrypted compression. As a result, as shown in Figure 14, we are able to consistently achieve more than 50% of the cross-snapshots storage cost reduction.

5.2.2 Impact of Page Fill Factor. To evaluate the impact of database page fill factor, we set the percentages of the tuples within each page that are randomly updated between two consecutive snapshots δ_p to be 10% and compare the storage cost reductions of the original page layout and the modified page layout with the page fill factor varying from 20%, 50%, to 70%.

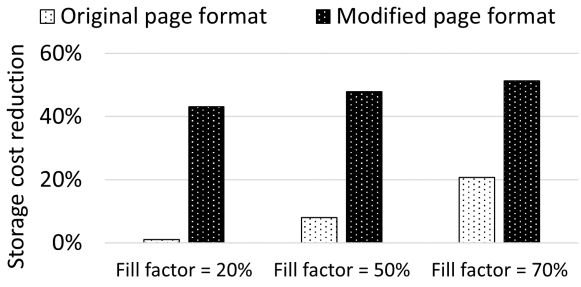

Figure 15: Snapshot storage cost reduction with the original and the modified page format with different fill factors.

As shown in Figure 15, as we increase the page fill factor, more data is written into the storage and the percentage of storage saving also increases. However, with the original page format, even if the fill factor increases to 70%, the storage cost reduction can only reach about 20%, because the header's metadata structure needs to be updated regardless of the fill factor or update intensity. In contrast, our optimized page structure can consistently achieve high storage cost reduction of over 40% under different fill factors, since our optimized page format moves the page header to the end and avoids inefficient intra-page in-place update.

6 Related Work

Secure data storage. Modern computing infrastructures emphasize not only processing capabilities but also stringent data security [7]. As data center systems evolve, secure cloud storage become increasingly important for cloud services, such as Microsoft Azure [34], Amazon Web Services (AWS) [6], Google Cloud [20] and IBM Cloud [26]. In order to protect sensitive information against cybersecurity threats, these providers often have a variety of rigorous measures implemented for data security. Central to these efforts is the deployment of advanced encryption protocols and strict access controls that ensure data integrity and confidentiality [33, 37, 42]. These platforms also employ comprehensive data management strategies, such as regular security audits and real-time threat detection, to maintain a secure environment [1, 11, 43].

Data compression and encryption accelerator. Data compression and encryption are computing-intensive tasks. Modern CPUs, such as Intel's Sapphire Rapids and IBM's Power9/z15 [2], feature on-chip compression accelerators. DPUs, such as Amazon's Nitro [8] and NVIDIA's Bluefield [13], embed compression and encryption accelerators, which effectively enhance performance in data center systems. Previous work on LPC [15] also made an effort to manage compressed and encrypted data on conventional storage with a compression utility. Our work, in contrast, leverages the new CSDs with an on-device, hardware-based compression engine and specially develops a CSD-based design for secure data management. Our study shows that CSD with transparent compression abilities plays a crucial role in achieving an efficient mapping structure between the host and the storage.

Computational storage drives. Computational storage drives have garnered high interest in recent research and are available in the commercial market. Notable examples include Samsung's SmartSSD [44], ScaleFlux's CSD [45], and DapuStor's products [25]. These developments are highlighted in several studies [10, 14, 30, 48], which explore the potential for database management systems to exploit computational SSDs with built-in transparent compression capabilities [17, 22–24, 51].

7 Conclusion

This paper demonstrates the great potential of adopting CSD technology for efficient data management in storage systems in modern data centers. By leveraging the unique properties of CSDs and carefully optimizing systems and application designs, our study shows that CSDs can enable numerous new opportunities to help us achieve the goal of building a well-balanced system. We hope that through the two highlighted cases, this work can inspire researchers and practitioners to further exploit opportunities with this new data storage technology to build secure, high-performance, and cost-efficient data center systems.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and insightful comments. This work was supported in part by the U.S. National Science Foundation under grants CCF-2210754, CCF-2312508, CCF-2517557, CCF-2517565, CCF-2210753, CCF-2312507, and OAC-2310510.

References

[1] Daniel J Abadi. 2009. Data management in the cloud: Limitations and opportunities.. In *IEEE Data Eng. Bull.*, Vol. 32. 3–12.
[2] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J Starke, Haren Myneni, and Charlie Wang. 2020. Data compression accelerator on IBM power9 and z15 processors: Industrial product. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.
[3] Ako Muhamad Abdullah. 2017. Advanced encryption standard (AES) algorithm to encrypt and decrypt data. In *Cryptography and Network Security*, Vol. 16. 11.
[4] A. Acharya, M. Uysal, and J. Saltz. 1998. Active disks: Programming model, algorithms and evaluation. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 81–91.
[5] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference* (Boston, Massachusetts) *(ATC'08)*. USENIX Association, USA, 57–70.
[6] Amazon Web Services. 2024. . Amazon. https://aws.amazon.com/
[7] VV Arutyunov. 2012. Cloud computing: Its history of development, modern state, and future considerations. In *Scientific and Technical Information Processing*. 173–178.
[8] AWS. 2022. *The security design of the AWS Nitro system*. Technical Report.
[9] AWS EBS. 2024. . Amazon. https://aws.amazon.com/ebs
[10] Antonio Barbalace and Jaeyoung Do. 2021. Computational storage: Where are we today?. In *Proc. of Annual Conference on Innovative Data Systems Research (CIDR)*.
[11] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, Simone Rondelli, Alexander Ryazanov, Michael Schmidt, Kunal Sengupta, Bryan Thompson, Diviji Vaidya, and Shawn Wang. 2018. Amazon Neptune: Graph data management in the cloud.. In *ISWC (P&D/Industry/BlueSky)*.
[12] BlobDB. 2024. . Meta. https://rocksdb.org/blog/2021/05/26/integrated-blob-db.html
[13] Idan Burstein. 2021. Nvidia data center processing unit (DPU) architecture. In *IEEE Hot Chips Symposium (HCS)*. 1–20.
[14] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *USENIX Conference on File and Storage Technologies (FAST)*. 29–41.
[15] Doron Chen, Michael Factor, Danny Harnik, Ronen Kat, and Eliad Tsfadia. 2021. Length preserving compression: marrying encryption with compression. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '21)*. Article 15, 12 pages.
[16] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems* (Seattle, WA, USA) *(SIGMETRICS '09)*. Association for Computing Machinery, New York, NY, USA, 181–192. doi:10.1145/1555349.1555371
[17] 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 *the Int'l Workshop on Data Management on New Hardware (DaMoN)*. 1–10.
[18] Morris J Dworkin. 2010. Recommendation for block cipher modes of operation: The XTS-AES mode for confidentiality on storage devices.
[19] Google. 2024. *LevelDB*. GitHub. https://github.com/google/leveldb
[20] Google Cloud Platform. 2024. . Google. https://cloud.google.com/
[21] Goetz Graefe. 2011. Modern B-tree techniques. In *Foundations and Trends® in Databases*, Vol. 3. 203–402.
[22] Kecheng Huang, Zhaoyan Shen, Zhiping Jia, Zili Shao, and Feng Chen. 2022. Removing double-logging with passive data persistence in LSM-tree based relational databases. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 101–116. https://www.usenix.org/conference/fast22/presentation/huang
[23] Kecheng Huang, Zhaoyan Shen, Zili Shao, Feng Chen, and Tong Zhang. 2025. HaSiS: A hardware-assisted single-index store for hybrid transactional and analytical processing. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 305–320. https://www.usenix.org/conference/fast25/presentation/huang
[24] Kecheng Huang, Zhaoyan Shen, Zili Shao, Tong Zhang, and Feng Chen. 2023. Breathing new life into an old tree: Resolving logging dilemma of B+-tree on modern computational storage drives. In *Proceedings of the VLDB Endowment*, Vol. 17. 134–147.
[25] Yunxin Huang, Aiguo Song, Chao Guo, and Yafei Yang. 2023. ASIC design of LZ77 compressor for computational storage drives. In *Electronics Letters*, Vol. 59. e13000.

[26] IBM Cloud. 2024. . IBM. https://www.ibm.com/cloud

[27] John Burkardt. 2024. *Data files.* Florida State University. https://people.math.sc.edu/Burkardt/data/data.html

[28] Julian McAuley. 2024. *Recommender systems and personalization datasets.* UCSD. https://cseweb.ucsd.edu/~jmcauley/datasets.html

[29] K. Keeton, D. Patterson, and J. Hellerstein. 1998. A case for intelligent disks (IDISKs). In *SIGMOD Rec.*, Vol. 27. 42–52.

[30] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. 2021. A fast and flexible hardware-based virtualization mechanism for computational storage devices. In *USENIX Annual Technical Conference (ATC)*. 729–743.

[31] Wilburt Labio and Hector Garcia-Molina. 1996. Efficient snapshot differential algorithms for data warehousing. In *Proceedings of the International Conference on Very Large Data Bases*.

[32] Moses Liskov, Ronald L Rivest, and David Wagner. 2002. Tweakable block ciphers. In *Advances in Cryptology—CRYPTO 2002: 22nd Annual International Cryptology Conference Santa Barbara, California, USA*. 31–46.

[33] Muhammad Sheraz Mehmood, Muhammad Rehman Shahid, Abid Jamil, Rehan Ashraf, Toqeer Mahmood, and Aatif Mehmood. 2019. A comprehensive literature review of data encryption techniques in cloud computing and IoT environment. In *2019 8th International Conference on Information and Communication Technologies (ICICT)*. 54–59.

[34] Microsoft Azure. 2024. . Microsoft. https://azure.microsoft.com/

[35] Oded Berman. 2020. *Snapshot copies: When to use them, when to tier them.* NetApp. https://www.netapp.com/blog/cts-blg-snapshot-copies-when-to-use-them-when-to-tier-them/

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). In *Acta Informatica*, Vol. 33. 351–385.

[37] Annanda Rath, Bojan Spasic, Nick Boucart, and Philippe Thiran. 2019. Security pattern for cloud SaaS: From system and data security to privacy case study in AWS and Azure. In *Computers*, Vol. 8. 34.

[38] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active storage for large-Scale data mining and multimedia. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*. 62–73.

[39] RocksDB. 2024. . Meta. https://rocksdb.org/

[40] Phillip Rogaway. 2004. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In *International Conference on the Theory and Application of Cryptology and Information Security*. 16–31.

[41] Mendel Rosenblum and John K Ousterhout. 1991. The design and implementation of a log-structured file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*. 1–15.

[42] Iman Saeed, Sarah Baras, and Hassan Hajjdiab. 2019. Security and privacy of AWS S3 and Azure Blob storage services. In *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 388–394.

[43] Sherif Sakr, Anna Liu, Daniel M Batista, and Mohammad Alomari. 2011. A survey of large scale data management approaches in cloud environments. In *IEEE communications surveys & tutorials*, Vol. 13. 311–336.

[44] Samsung SmartSSD. 2024. . Samsung. https://samsungmsl.com/smartssd2/

[45] ScaleFlux CSD. 2024. . ScaleFlux. https://scaleflux.com/

[46] Silesia Corpus. 2024. . Github. https://github.com/MiloszKrajewski/SilesiaCorpus

[47] Dejun Teng, Lei Guo, Rubao Lee, Feng Chen, Siyuan Ma, Yanfeng Zhang, and Xiaodong Zhang. 2017. LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 68–79. doi:10.1109/ICDCS.2017.70

[48] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under HTAP workloads. In *Proceedings of the VLDB Endowment*, Vol. 15. 1991–2004.

[49] Wiredtiger. 2024. . Github. https://github.com/wiredtiger/wiredtiger

[50] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A comprehensive study of the past, present, and future of data deduplication. In *Proceedings of the IEEE*, Vol. 104. 1681–1710.

[51] Ning Zheng, Xubin Chen, Jiangpeng Li, Qi Wu, Yang Liu, Yong Peng, Fei Sun, Hao Zhong, and Tong Zhang. 2020. Re-think data management software design upon the arrival of storage hardware with built-in transparent compression. In *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.