Re-think Data Management Software Design Upon the Arrival of Storage Hardware with Built-in Transparent Compression

Ning Zheng *ScaleFlux Inc.*
Xubin Chen *RPI*
Jiangpeng Li *ScaleFlux Inc.*
Qi Wu *ScaleFlux Inc.*
Yang Liu *ScaleFlux Inc.*
Yong Peng *ScaleFlux Inc.*
Fei Sun *ScaleFlux Inc.*
Hao Zhong *ScaleFlux Inc.*
Tong Zhang *ScaleFlux Inc. and RPI*

Abstract

This position paper advocates that storage hardware with built-in transparent compression brings new opportunities to innovate data storage management software (e.g., database and filesystem). Modern storage appliances (e.g., all-flash array) and some latest SSDs (solid-state drives) can perform data compression transparently from OS and user applications. Such storage hardware decouples logical storage space utilization efficiency from physical storage space utilization efficiency. This allows data storage management software intentionally waste logical storage space in return for employing simpler data structures, leading to lower implementation complexity and higher performance. Following this theme, we carried out three preliminary case studies in the context of relational database and key-value (KV) store. Initial experimental results well demonstrate the promising potential, and it is our hope that this preliminary study will attract more interest towards exploring this new research area.

1 Introduction

This position paper advocates that storage hardware with built-in transparent compression brings exciting opportunities to innovate data management software (e.g., database and filesystem). Commercial market has witnessed the rise of block storage appliances/devices that perform data compression with complete transparency to OS and user applications. Modern all-flash arrays (e.g., Dell EMC PowerMAX [1], HPE Nimble Storage [2], and Pure Storage FlashBlade [4]) support block-level transparent compression. SSDs with built-in transparent compression are also emerging on the market (e.g., computational storage drive from ScaleFlux [5] and Nytro SSD from Seagate [13]).

In addition to its apparent benefit on reducing the storage cost, storage hardware with built-in transparent compression **decouples** the logical storage space utilization efficiency from the physical storage space utilization efficiency. This creates a new spectrum for data management software innovation, which can be explained as follows. When running on conventional storage hardware, data management software is *solely responsible* for the physical storage space utilization efficiency. As a result, data management software faces a stringent trade-off between storage utilization efficiency and implementation complexity: In order to improve the storage space utilization, data management software should make full use of the logical storage space and completely fill each 4KB LBA (logical block address) sector with user data, which demands more sophisticated data structures and algorithms. In comparison, when running on storage hardware with built-in transparent compression, data management software can *purposely waste* the logical storage space to reduce its implementation complexity, while relying on the storage hardware to retain physical storage space efficiency. Lower software implementation complexity may lead to higher speed performance and better system stability.

Little prior research has studied how data management software can effectively leverage the decoupled logical vs. physical storage utilization efficiency. Regardless of specific application, the essential design theme is that data management software judiciously *wastes* the logical storage space in return for lower implementation complexity and higher performance. As storage hardware with built-in transparent compression emerges on the mainstream market, it becomes increasingly necessary to re-think the data management software design under this framework. As the first effort along this direction, we carried out three preliminary case studies: (1) We show that PostgreSQL (the second most popular open-source relational database) can easily benefit from this theme by simply adjusting a parameter; (2) We show that log-structured data store can leverage this theme to reduce the impact of background GC (garbage collection); (3) We show that one could apply this theme to implement a hash-based KV store at almost zero memory usage. It is our hope that these preliminary case studies will contribute to attracting more research interest and activities towards this largely unexplored territory, which may lead to unforeseen opportunities to innovate future data management software.

2 Proposed Design Theme

For systems running on conventional storage hardware, their logical and physical storage space utilization always tightly couple together, i.e., the LBA space is (almost) equal to the physical storage space inside the storage hardware, and each 4KB LBA sector always occupies 4KB physical storage space. Therefore, data management software is solely responsible for the physical storage space utilization efficiency. As a result, data management software typically employs sophisticated data structures (e.g., B-tree and log-structured merge tree) and algorithms in order to fully utilize the physical storage space. This however leads to high implementation complexity, high CPU/memory resource usage, and difficulty on achieving high speed and stability. A majority of prior research on data management systems focused on searching for better design trade-offs between storage cost and implementation/performance cost.

For the purpose of illustration, Fig. 1 shows the structure of an SSD with built-in transparent compression. Data compression and decompression are carried out on the IO path by the hardware engine inside the SSD controller, being transparent to the host software stack. The FTL (flash translation layer) inside SSD controller manages the mapping/indexing of all the variable-length compressed data blocks. By compressing each LBA sector and exposing a LBA space much larger than the physical storage space, storage hardware with built-in transparent compression decouples the logical storage space utilization efficiency from the physical storage space utilization efficiency. This allows data management software to purposely under-utilize the logical storage space in return for using less sophisticated data structures and algorithms. This can lead to lower implementation complexity, less CPU/memory resource usage, and higher performance and stability. Accordingly, we propose to re-think the design of data management software from two aspects:

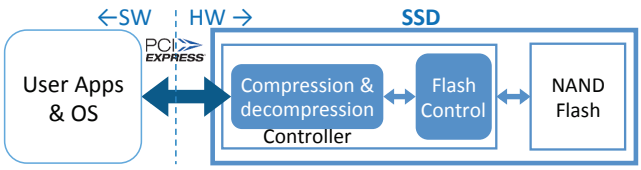

Figure 1: Illustration of an SSD with built-in transparent compression.

1. *Under-utilize LBA space*: Storage hardware with built-in transparent compression can natively expose an LBA space that is much larger than the physical storage space. We should investigate whether data management software can employ simpler data structure and algorithm by intentionally wasting the abundant LBA space.
2. *Under-utilize each LBA*: We note that special data patterns (e.g., all-zero and all-one vectors) can be highly compressed even with simple compression algorithms such as lz4 and Snappy. Hence, we should investigate whether data management software can employ simpler data structure and algorithm by intentionally wasting the 4KB storage space of each LBA (i.e., leave each 4KB LBA sector partially filled with user data and padded with all-zero vectors).

3 Preliminary Case Studies

Following the theme presented above, we carried out three case studies that apply the proposed theme to (1) improve the performance of PostgreSQL at minimal storage space overhead, (2) reduce the impact of GC in log-structured data store, and (3) architect a new highly efficient KV store. All the experiments were carried out based on commercial SSDs [5] that support built-in transparent compression and achieve the same IOPS (IO per second) and throughput performance as leading-edge normal NVMe SSDs.

3.1 Improve the Performance of PostgreSQL

3.1.1 Background

PostgreSQL applies B-tree index to manages its data storage, and realizes MVCC (multi-version concurrency control) by storing all the row versions in the table space. Hence, instead of directly in-place updating a row, PostgreSQL always first stores the new row version at a new location and relies on a background *vacuum* process to reclaim the table space occupied by dead row versions. As a result, the performance of updating non-index fields in a row strongly depends on whether PostgreSQL can store the new row version in the same page as the old row version:

- If the page hosting the old row version is full, PostgreSQL has to store the new row version in another page. Hence, PostgreSQL must accordingly modify the B-tree structure by manipulating (and splitting or creating) one or multiple additional pages. This causes extra CPU usage and performance degradation.
- If the page hosting the old row version has sufficient empty space, PostgreSQL simply appends the new row version in that page. By keeping the B-tree structure intact, this causes very low CPU usage, leading to a higher speed performance.

The above fact reveals a fundamental trade-off between TPS (transactions per second) performance and storage space usage: When inserting new rows into a page, if we do not completely fill the page and reserve the remaining empty space to absorb future updates, we can improve the TPS performance. Nevertheless, this meanwhile leads to larger storage space usage. PostgreSQL allows users to configure such a trade-off by exposing a parameter called *fillfactor*. Being a percentage value between 10 and 100, it controls how full

each page will be filled with inserted rows. Its default value is 100, i.e., each page is 100% filled with inserted rows and hence does not reserve any space for future updates.

3.1.2 Basic Concept and Experimental Results

Storage hardware with built-in transparent compression makes PostgreSQL much less subject to the above performance vs. storage cost trade-off, leading to a unique opportunity to improve the PostgreSQL TPS performance at minimal storage cost. As pointed out above in Section 2, special data patterns like all-zeros can be highly compressed. Meanwhile, with *fillfactor* being less than 100, PostgreSQL initializes the reserved space in each page as zeros. Hence, when running PostgreSQL on SSDs with built-in transparent compression, the all-zero segment in each page only occupies very small amount of physical flash memory storage space. Therefore, by decoupling the logical vs. physical storage space utilization efficiency, storage hardware with built-in transparent compression allows PostgreSQL aggressively reduce the *fillfactor* to improve the TPS performance at very small increase of physical storage space usage.

To further demonstrate this concept, we carried out experiments on PostgreSQL (version 10.10) using the Percona Sysbench-TPCC OLTP benchmark [3]. We used a server with 32-core 3.3GHz Xeon CPU and 64 client threads. For the purpose of comparison, we run the same experiments on one 3.2TB NVMe SSD and one 3.2TB SSD with built-in transparent compression. By keeping the *fillfactor* as its default value of 100, the PostgreSQL TPS is 3,214 and physical storage usage is 740GB in the case of NVMe SSD, and the PostgreSQL TPS is 3,128 and physical storage usage is 178GB (i.e., the Sysbench dataset can be compressed from 740GB to 178GB) in the case of SSD with built-in transparent compression. By reducing the *fillfactor* to 75, the PostgreSQL TPS improves to 4,265 and physical storage usage jumps to 905GB in the case of NVMe SSD, and the PostgreSQL TPS improves to 4,330 and physical storage usage slightly increases to 198GB in the case of SSD with built-in transparent compression. As further illustrated in Fig. 2, the results suggest that, by simply configuring the *fillfactor* parameter, PostgreSQL can noticeably benefit from the decoupled logical vs. physical storage space utilization efficiency.

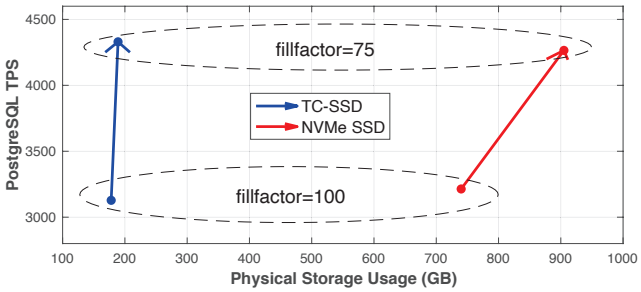

Figure 2: Illustration of the measured Sysbench-TPCC TPS performance and physical storage usage, where TC-SSD means SSD with built-in transparent compression.

3.2 Reduce the Impact of GC

3.2.1 Background

Log-structured design principle [21, 27] has been widely used to implement modern data management systems, e.g., SSD-oriented file systems [15, 31]) and KV stores [12, 18, 24, 26]. Because log-structured data stores do not perform in-place updates, stale data will accumulate over the time, leading to the data storage *bloating* (or space amplification). To limit the storage space amplification, storage systems must periodically carry out background GC operations to reclaim the storage space at the cost of write amplification. This leads to a fundamental trade-off between space amplification and write amplification: Let C_{val} and C_{inval} denote the amount of valid and invalid data in the log-structured data store. Define $\gamma = (C_{val} + C_{inval})/C_{val}$ as the data bloating factor, and we trigger GC whenever γ is larger than a given threshold γ_{th}. As we reduce the threshold γ_{th}, the runtime data bloating will become less significant (i.e., the peak space amplification will reduce), and meanwhile GC-induced write amplification will increase and hence GC will more noticeably degrade the system performance.

3.2.2 Basic Design Concept

Following the proposed theme, we present a method called *virtual data trim* to make log-structured data store less subject to the space amplification vs. write amplification trade-off. As illustrated in Fig. 3, the basic concept is to reset the content of invalid data elements to all-zeros, through which we can rely on the transparent compression, instead of GC, to reclaim the physical storage space occupied by invalid data. It only reduces the physical storage space bloating, and the logical storage space bloating remains unchanged. Therefore, once we schedule GC based on the physical storage space bloating, we can reduce the frequency of GC operations, which can reduce the GC-induced write amplification and hence reduce its impact to the system performance.

We note that virtual data trim also induces write amplification, since it carries out read-modify-write in order to reset the data content. Therefore, we should perform virtual data trim only if it causes sufficiently small write amplification. Given the 4KB SSD sector size, we propose the following strategy to implement the virtual data trim: Within each 4KB sector, let S_{val} denote the amount of valid data and define $\gamma^{(s)} = 4KB/S_{val}$. We perform virtual data trim only for those sectors whose γ_s is larger than the threshold γ_{th}. We note that, if S_{val} is 0 (i.e., the 4KB sector contains only invalid data), we can simply trim the entire sector, instead of using virtual data trim. Accordingly, we should modify the GC schedul-

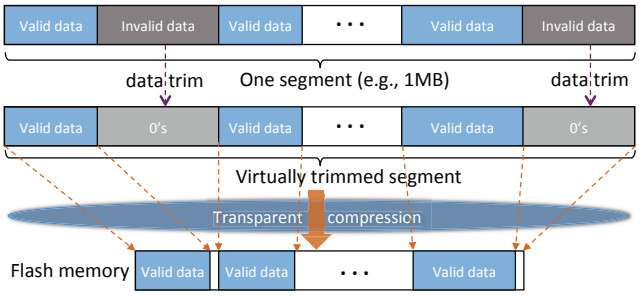

Figure 3: Illustration of the proposed virtual data trim design concept to reclaim physical storage space without GC.

ing as follows: Let $C_{val}^{(l)}$ and $C_{inval}^{(l)}$ denote the logical storage space occupied by the valid data and invalid data, and $C_{trim}^{(l)}$ denote the amount of invalid data that have been trimmed. We define the *adjusted* bloating factor

$$\gamma_t = \frac{C_{val}^{(l)} + C_{inval}^{(l)} - C_{trim}^{(l)}}{C_{val}^{(l)}}, \quad (1)$$

and trigger GC whenever γ_t is larger than the threshold γ_{th}.

3.2.3 Preliminary Experimental Results

To preliminary evaluate the proposed design approach, we implement a simple log-structured data store that consists of a number of 64MB segments. At one time, only one segment is open to receive inserted/updated data elements in the append-only manner. Once the open segment reaches 64MB, it will be closed, and a new empty segment will be opened. During each GC operation, we search among all the closed segments and pick the ones with the highest garbage rate for recycling. For the purpose of comparison, we studied two scenarios: (1) *Current practice*: Following the current practice of log-structured data store design, all the closed segments are strictly immutable, and we do not apply data trim; (2) *Trim-assisted*: We apply the proposed virtual data trim to runtime reclaim the physical storage space, where all the segments are no longer strictly immutable. We set each data element is 2KB, and Table 1 lists the measured write amplification (WA) and physical space amplification (PSA). The results show that the proposed design approach indeed could enable a more favorable trade-off between write amplification and physical space amplification.

Table 1: Measured write and physical space amplification.

	Current practice			Trim-assisted
γ_{th}	1.2	1.3	1.4	1.2
WA	3.20	2.34	1.93	2.16
PSA	120%	131%	141%	119%

3.3 Hash-based KV Store

3.3.1 Background

To implement a KV store, the core design decision is the index data structure, which can be either tree-based or hash-based. However, compared with significant prior efforts on tree-based KV store (e.g., see [7, 10, 12, 18, 22, 24, 30]), little prior efforts chose to focus on hash-based KV store [11], even though hash-based approach can support higher index access throughput. The in-memory data store *Redis* [25] appears to be the only commercially successful hash-based KV store. Arguably, this is mainly due to the very high memory cost of hash-based approach, especially compared with the ones built upon log-structured merge tree.

In conventional practice, hash-based KV store must use an in-memory hash table to maintain the mapping from the key space to storage space. Such *indirect* addressing through the intermediate hash table can ensure the compact placement of KV pairs on the storage space and hence maximize the storage space utilization. Meanwhile, the memory footprint of hash table is directly proportional to the number of KV pairs, leading to prohibitively high memory cost for large-scale hash-based KV stores.

3.3.2 Basic Design Concept

Following the theme of leveraging the decoupled logical vs. physical storage utilization efficiency, we propose a *table-less hash-based* KV store design approach as illustrated in Fig. 4. The basic idea is to directly hash the key space onto the logical storage space, without going through an intermediate hash table. In particular, let $\mathbb{K}$ denote the key space of the KV store and $\mathbb{L}$ denote the logical LBA storage space. We use a hash function $f_{K\to L}$ to hash each key $K_i \in \mathbb{K}$ onto one LBA $L_j \in \mathbb{L}$. By obviating the use of a hash table, it eliminates the memory cost obstacle faced by conventional implementation of hash-based KV store. Moreover, it relieves CPU from managing/searching hash table, leading to less CPU usage. As pointed out above, indirect addressing through an in-memory hash table can ensure the compact placement of KV pairs on the logical storage space. In contrast, as illustrated in Fig. 4, the proposed approach is fundamentally subject to logical storage space under-utilization (i.e., almost all the LBAs have empty space left unoccupied). Once we keep the content of the unoccupied space as all-zeros, storage hardware with built-in compression can naturally retain the physical storage under-utilization.

Converting this simple design approach into a commercially viable KV store certainly is nontrivial, and must adequately address several open issues and overcome many engineering challenges. For example, we should effectively handle the occurrence of hashing overflow when more than 4KB of KV pairs are hashed onto the same LBA. In this case, we should use a separate data store to host those spilled-

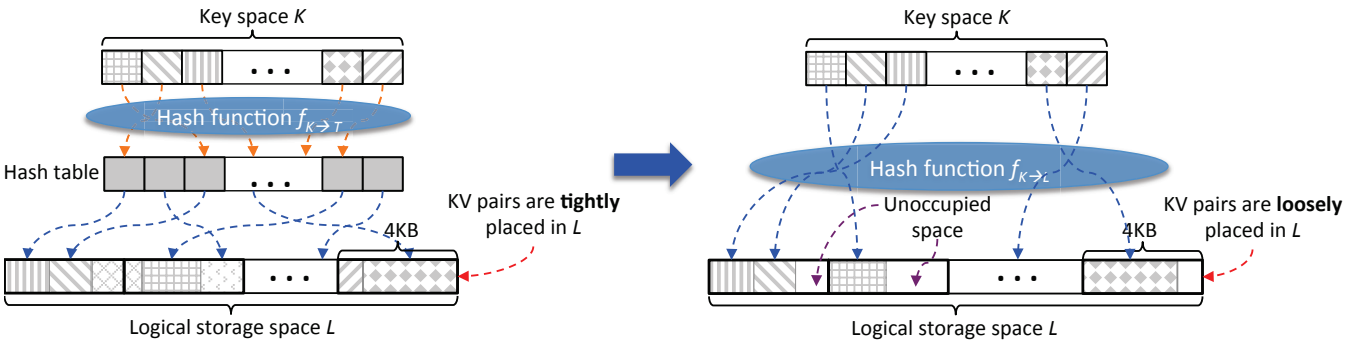

Figure 4: Illustration of the proposed table-less hash-based KV store design approach.

over KV pairs. Moreover, given the key space $\mathbb{K}$, we must make the LBA space (i.e., $|\mathbb{L}|$) large enough in order to make the hashing overflow rate sufficiently low (e.g., below 1%). Therefore, as the size of the key space varies, we must accordingly adjust the size of the LBA space and meanwhile ensure minimal impact on the runtime KV store performance. It is also highly desirable to develop a mathematical formulation framework that can guide the runtime LBA space re-sizing.

3.3.3 Preliminary Experimental Results

We implemented a preliminary KV store prototype that can realize the basic GET and PUT operations, and use embedded SQLite [29] to absorb the spilled-over KV pairs. The KV store serves all the PUT requests in an asynchronous manner, i.e., after recording the PUT requests in the write-ahead log (WAL), client threads submit PUT requests to a queue, and background threads process the queue in a batch mode with asynchronous IOs. As pointed out in [16], the use of asynchronous IOs can very effectively improve the throughput at low CPU usage. To improve the operational parallelism, we partition one table space into multiple sections, each section has its own PUT request queue and associates with one background thread that processes the PUT request queue. All the GET requests are served by client threads through synchronous direct-IOs. For the purpose of evaluation, we carried out experiments on this KV store prototype and RocksDB with the following configurations: We loaded 2 billion KV pairs, where each key and value is 16-byte and 400-byte, respectively. We kept the default RocksDB settings, and the background thread pool size is 32. For our KV store prototype, we partitioned the table space into 16 sections. Table 2 lists the measured results when running 32 client threads with 7:3 GET vs. PUT ratio. The results shows that the table-less hash-based KV store has a very promising potential to achieve significantly higher performance and meanwhile consume less CPU cycles and almost zero memory capacity, compared with the popular RocksDB.

Table 2: Measured KV store performance and CPU usage.

	ops/s	Read latency		CPU utilization
		Avg.	99%	
Proposed	275K	130μs	605μs	21.7%
RocksDB	182K	240μs	915μs	43.7%

4 Related Work

Research community has long studied the benefit and trade-off of applying compression in database [6, 19, 28, 14, 23], and investigated the implementation of transparent compression at the filesystem level [9, 8] and block device level [20, 17]. By implementing an emulator for SSD with built-in transparent compression, Zuck *et al.* [32] studied the options of integrating transparent compression into SSD, and demonstrated its potential of reducing storage cost for relational database without sacrificing TPS performance. Most prior work studied the use of compression solely for the purpose of reducing the data storage cost. In comparison, this work studies the potential of simplifying data management software in the presence of storage hardware with built-in transparent compression.

5 Conclusion

This position paper for the first time points out that, by decoupling the logical storage space utilization from the physical storage space utilization, storage hardware with built-in transparent compression enables a promising potential to innovate data storage management software. The essential theme is to make data storage management software intentionally and appropriately under-utilize the logical storage space in return for employing simpler data structures and algorithms, which can further enable lower implementation complexity and higher performance. We carried out three preliminary case studies that apply this design theme in the context of relational database and KV store, and the results well demonstrate the promise.

6 Discussion

Given the wide landscape of data management software ecosystem, there will be numerous open questions and unforeseen opportunities as the research community start to explore this proposed direction. Here we list several open questions which hopefully may serve as a catalyst.

Application to more relational databases: This position paper demonstrates how PostgreSQL may benefit from the proposed theme by simply configuring a parameter. It remains unclear how other popular relational databases (e.g., MySQL and Oracle) could utilize this theme. Because they implement MVCC using different strategies than PostgreSQL, we expect that one may need to appropriately modify their source code in order to gain similar benefits.

Integration into log-structured data store: This position paper presents very preliminary results on applying virtual data trim to reduce the GC impact for log-structured data store. Much more research is needed to fully understand the trade-offs and study how to practically integrate the proposed technique into real-world log-structured data stores.

Implementation of table-less hash-based KV store: This position paper outlines the basic idea of the proposed table-less hash-based KV store. By eliminating the memory-hungry hash table, it has a promising potential to enable high-performance, low-cost alternatives to existing KV stores such as RocksDB. Of course, there are many open issues to be addressed, e.g., how to most effectively handle hashing overflow, how to realize LBA space resizing at the minimal performance impact, and how it can support additional features such as snapshot and transaction.

Application to data analytics: Data analytics typically employ column-store to improve the performance and reduce the storage cost. However, the heavy use of compression in today's column-store makes it almost impossible to effectively serve transactional queries. Column-stores may possibly leverage the decoupled logical vs. physical storage space utilization to mitigate this issue, which still remains a completely open question.

Application to filesystems: As the core operation of any filesystems, storage space allocation and indexing involve a trade-off between storage space utilization efficiency and implementation complexity (and performance). This clearly leads to a potential of leveraging the proposed theme in the context of filesystems.

Application to SSD-based caching for HDDs: Complementing HDDs with SSD-based cache can improve the storage system performance at modest cost overhead. In conventional practice, cache must maintain a complicated index data structure to manage the mapping between HDD storage space and SSD-based cache space. Following the proposed design theme, one could envision a cache design solution that obviates the explicit use of indexing, which can reduce the implementation complexity of the SSD-based cache.

References

[1] *Dell EMC PowerMax*. https://delltechnologies.com/.

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

[3] *Percona Sysbench-TPCC*. https://github.com/Percona-Lab/sysbench-tpcc.

[4] *Pure Storage FlashBlade*. https://purestorage.com/.

[5] *ScaleFlux Computational Storage*. http://scaleflux.com.

[6] ALSBERG, P. A. Space and time savings through large data base compression and dynamic restructuring. *Proceedings of the IEEE 63*, 8 (1975), 1114–1122.

[7] BALMAU, O., DIDONA, D., GUERRAOUI, R., ZWAENEPOEL, W., YUAN, H., ARORA, A., GUPTA, K., AND KONKA, P. TRIAD: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2017), pp. 363–375.

[8] BURROWS, M., JERIAN, C., LAMPSON, B., AND MANN, T. On-line data compression in a log-structured file system. *ACM SIGPLAN Notices 27*, 9 (1992), 2–9.

[9] CATE, V., AND GROSS, T. Combining the concepts of compression and caching for a two-level filesystem. *ACM SIGOPS Operating Systems Review 25*, Special Issue (1991), 200–211.

[10] DAYAN, N., AND IDREOS, S. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2018), ACM, pp. 505–520.

[11] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the ACM International Conference on Management of data (SIGMOD)* (2011), pp. 25–36.

[12] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing Space Amplification in RocksDB. In *CIDR* (2017), vol. 3, p. 3.

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

[14] IYER, B. R., AND WILHITE, D. Data compression support in databases. In *VLDB* (1994), vol. 94, pp. 695–704.

[15] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A new file system for flash storage. In *USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[16] LEPERS, B., BALMAU, O., GUPTA, K., AND ZWAENEPOEL, W. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2019), pp. 447–461.

[17] LI, C., SHILANE, P., DOUGLIS, F., SHIM, H., SMALDONE, S., AND WALLACE, G. Nitro: A capacity-optimized SSD cache for primary storage. In *USENIX Annual Technical Conference (ATC)* (2014), pp. 501–512.

[18] LU, L., PILLAI, T. S., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS) 13*, 1 (2017), 5.

[19] LYNCH, C. A., AND BROWNRIGG, E. B. Application of data compression to a large bibliographic data base. In *Proceedings of the International Conference on Very Large Data Bases* (1981), pp. 435–447.

[20] MAKATOS, T., KLONATOS, Y., MARAZAKIS, M., FLOURIS, M. D., AND BILAS, A. Using transparent compression to improve SSD-based I/O caches. In *Proceedings of the European conference on Computer systems* (2010), pp. 1–14.

[21] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[22] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of USENIX Annual Technical Conference (ATC)* (2016), pp. 537–550.

[23] POESS, M., AND POTAPOV, D. Data compression in oracle. In *Proceedings of VLDB Conference* (2003), Elsevier, pp. 937–947.

[24] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2017), pp. 497–514.

[25] REDIS. https://redis.io.

[26] ROCKSDB. https://github.com/facebook/rocksdb.

[27] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[28] SEVERANCE, D. G. A practitioner's guide to data base compression tutorial. *Information Systems 8*, 1 (1983), 51–62.

[29] SQLITE. https://www.sqlite.org/.

[30] YUE, Y., HE, B., LI, Y., AND WANG, W. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems 28*, 4 (2016), 961–973.

[31] ZHANG, J., SHU, J., AND LU, Y. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *USENIX Annual Technical Conference (ACT)* (2016), pp. 87–100.

[32] ZUCK, A., TOLEDO, S., SOTNIKOV, D., AND HARNIK, D. Compression and SSDs: Where and how? In *Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)* (2014).