Leveraging Keys In Key-Value SSD for Production Workloads

Manoj P. Saha
Florida International University
Miami, Florida, USA
msaha002@fiu.edu

Omkar Desai
Syracuse University
Syracuse, New York, USA

Bryan S. Kim
Syracuse University
Syracuse, New York, USA

Janki Bhimani
Florida International University
Miami, Florida, USA

ABSTRACT

Key-Value SSDs reduce host-side resource utilization for unstructured data management by streamlining the I/O stack. However, designing a robust Key-Value SSD with resource constrained flash controllers has always been a challenge. The key-to-page (K2P) mapping inside KV-SSD, which consolidates multiple layers of indirection in the traditional block I/O storage, has its own shortcomings. The sparsely populated NVMe KV namespace leads to very large index, which cannot be optimized similar to hybrid- or block-FTL in block-SSDs. In addition, the background index management tasks (e.g. compaction on LSM-tree index) also lead to performance degradation. Moreover, existing KV index design is not equipped to tackle fast changing workload patterns. These shortcomings have stalled the adoption of KV-SSDs in production environments. In this work, we take the position that these shortcomings can be addressed by leveraging the information embedded inside keys about application keyspaces and groups as prefixes. The prefixes can be used to partition the monolithic large index into smaller ones. We demonstrate a naive prefix-based index partitioning mechanism inside KV-SSD that can reduce on-flash index accesses for multiple production workloads and discuss the shortcomings of this approach. Lastly, we discuss our proposed design of a society of indices that initialize, interact and evolve based on workload characteristics over time.

CCS CONCEPTS

• **Information systems** → **Flash memory**.

KEYWORDS

Key-Value SSD, Data Storage, KV Indexing, Key Prefix

ACM Reference Format:
Manoj P. Saha, Omkar Desai, Bryan S. Kim, and Janki Bhimani. 2023. Leveraging Keys In Key-Value SSD for Production Workloads . In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23), June 16–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3588195.3595949

HPDC '23, June 16–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0155-9/23/06.
https://doi.org/10.1145/3588195.3595949

1 MOTIVATION

Key-Value SSDs have reduced system resources required for managing unstructured data by streamlining the I/O stack and offloading the tasks of data management to the SSD [4]. However, it is yet to see wide adoption in production environments due to the following performance bottlenecks [3].

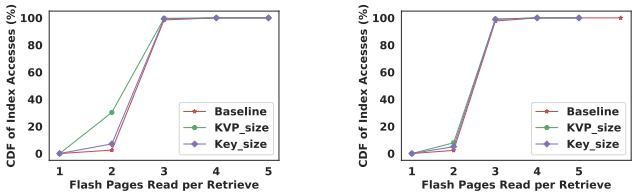
The KV index size can be enormous. The average KV pair size in many production workloads [1, 5] are small in size (from less than 100B to a few KB). The keys are also sparse. A single SSD stores billions of keys, making the KV index size huge. For example, storing 4TB data with 32B keys and 1KB values will result in a 144GB KV index. However, the 4GB SSD DRAM (1% of SSD capacity) can only cache a fraction of the index [3]. This leads to frequent flash accesses to fetch translation pages (i.e., NAND flash pages storing the index) and reduces I/O performance. The enormous size of index also increases index management overheads (e.g., compaction of translation pages in a LSM-tree index).

Index cannot adapt to changing Workloads. Changes in workload characteristics can increase background index management overheads and degrade foreground I/O performance. For example, a sudden influx of updates can lead to increased garbage collection or index compaction (in case of LSM-tree index) of translation pages. A monolithic and large global index with predefined index management policies (e.g. compaction policy in LSM-tree index) may not be able to adapt to the changing workloads fast enough.

2 CORE INSIGHTS

Breaking down a large index into smaller ones can reduce index access delays. Index partitioning is a common technique used by databases to improve performance. However, the SSD is oblivious to the information needed to realize efficient index partitioning. The application-side data management abstractions are inherently different from data management abstractions in the NVMe layer. The question then becomes - how can we partition the KV index? We deploy a simple index partitioning scheme based on key size and KV pair size. The resulting smaller indices (for both LSM-tree index and multi-level hash index) incur smaller search paths when retrieving mapping entries from the index. This leads to lower flash reads for index access for the smaller indices (Fig. 1). In addition, the compaction and garbage collection policies for each index can be set separately to optimize these operations. However, these partition schemes cannot adapt to changing workload patterns.

Although different key-value applications follow different data management abstractions and hierarchies, all of this information is added to the keys as prefixes. KV applications



(a) Twitter Trace 28. (b) Twitter Trace 52.

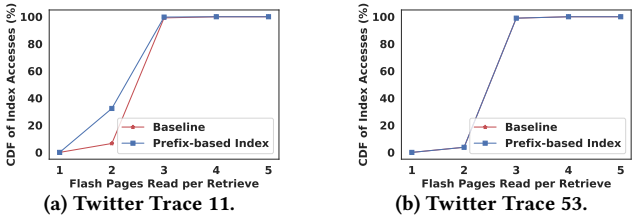
Figure 1: NAND flash page accesses to retrieve mapping entries from flash-resident index.

use a myriad of abstractions to separate and manage keys in groups. These abstractions include keyspace (KS), keygroup (KG), column family (CF), bucket etc. Oftentimes, a single application uses multiple of these abstractions together. For example, CF1:KS1:...:KEYID. Regardless of the abstractions and their hierarchies used by KV applications, this information is added to keys as prefixes. Since these prefixes are added in hierarchical order in the keys, the data management hierarchy can be extracted easily inside KV-SSD from the keys, without prior knowledge of the application. This information can then be utilized for index partitioning inside KV-SSD.

3 NAIVE PREFIX-BASED INDEXING

Figure 2: Naive prefix-based indexing.

Based on the core insights, we implement a naive prefix-based indexing mechanism (Fig. 2). We use a delimiter-based prefix identifier that uses only the top level of any application's internal data management hierarchy. We initialize indices for all extracted prefixes. A prefix table stores the prefix and index pointer pairs. Each index works independently and is unaware of the others. We evaluate the merits of the design using extensible multi-level hash indices [2]. Preliminary results show that the naive approach reduces average flash accesses required for retrieving keys (i.e. mapping entries) (Fig. 3a). The overall I/O latency reduces accordingly.



(a) Twitter Trace 11. (b) Twitter Trace 53.

Figure 3: NAND flash page accesses to retrieve mapping entries from flash-resident index with prefix-based indexing.

However, the naive prefix-based indexing approach has three problems. First, it cannot improve the performance of workloads with a single top-level prefix (Fig. 3b). This can be mitigated by using second- or third-level prefixes. For example, Twitter trace 53 has one top-level prefix and six second-level prefixes (Table 1). However, some workloads have an arbitrarily large number of top-level or second-level prefixes. Managing so many prefixes/indices inside the resource-constrained KV-SSD is impractical. For example, Memtables of all LSM-tree-based indices need to be kept in the SSD DRAM. Given the limited SSD DRAM size, keeping 71437 Memtables (Trace 49) in SSD DRAM is not possible. Lastly, the naive prefix-based indexing approach does not offer any inherent mechanism to adapt to changing workload patterns.

Trace	Top-level Keygroups	Second-level Keygroups
11	6	6
49	71437	95819
53	1	6

Table 1: Data management hierarchy in different workloads.

4 A SOCIETY OF INDICES

Armed with our findings in Section 3, we propose our design of "a society of indices" that initialize, interact and evolve based on workload characteristics over time (Fig. 4). In this approach, we propose a combination of a global index and one or more prefix-based indices to improve I/O performance. We replace the multi-level hash indices with LSM-tree indices. We also redesign our prefix extraction component to extract both prefixes and their hierarchy. A lightweight prefix-based I/O profiler keeps track of accesses to each prefix. Only if a particular prefix has a significantly high number of accesses, a new index will be initialized for that prefix, which will work as the primary index for the same. The global index will work as a secondary index for that prefix to store "cold" mapping entries. A novel joint compaction process will merge cold SSTable files from prefix-based index to the global index. In this way, the prefix-based primary indices remain lightweight. In addition, policies to manage each prefix-based index will be periodically updated based on the I/O profiler data. We plan to implement the new design inside a QEMU-based KV Emulator and evaluate it using real-world production workload traces.

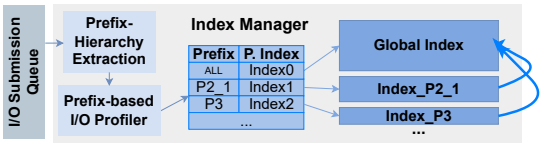


Figure 4: A society of indices.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation (NSF) Awards CNS-2008324 and CNS-2122987, and KV-SSD equipment grant and funded research collaboration with Samsung MSL GRO (Global Research Outreach).

REFERENCES

[1] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[2] Steven Feldman, Pierre LaBorde, and Damian Dechev. 2013. Concurrent multi-level arrays: Wait-free extensible hash maps. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 155–163. https://doi.org/10.1109/SAMOS.2013.6621118

[3] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 173–187. https://www.usenix.org/conference/atc20/presentation/im

[4] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards Building a High-Performance, Scale-in Key-Value Storage System. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/3319647.3325831

[5] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang