Minimizing Compaction Impact via Unified Data Format for LSM

Jinghuan YU, Xuan Sun, Chun Jason Xue, Cheng Ji
Department of Computer Science, City University of Hong Kong

Abstract

Combining NVM (non-volatile memory) with LSM(log-structured-merged tree) has received extensive attention recently, researchers are interested in how can these byte-addressing materials help to improve LSM's weaknesses. Recent solutions mostly focus on how to access faster for one part of the LSM's components while we evaluate there are still gaps between different storage components. Inspired by the evaluation upon conversion costs for SST (Sorted String Table) files to different memory processable data block formats, we design a unified data format for JigsawDB to achieve light overhead data conversion. We describe the prototype of JigsawDB, a single storage LSM-based system, which can reduce the overhead of data conversion among different storage systems by unified data blocks. This united format produces much less overhead than traditional transformation process. At last, we evaluate the JigsawDB's performance against LevelDB and its hierarchical architecture equipped with NVM. As a result, JigsawDB accelerates 28.1% for overall execution in the hybrid storage system while hierarchical solution only improves 17.19%.

1 Introduction

LSM Tree is a high warm writing performance data structure proposed in 1996 [17], gets widely used in many products like Cassandra [1], Hbase [2], BigTable [9] and WiredTiger [7]. LSM uses incremental changes to optimize write throughput, converts random write operations into sequential accessing. However, this introduces extra overheads in space cost and garbage collection. Moreover, when system's persistent component (the stored files) grows larger and larger, the query performance becomes relatively low, repeated records have not been updated in-time will remain in files will lead to further waste of storage space. The additional collection operations (also known as Compaction) applied to these data will introduce further space amplification. Many studies have been researched to improve this problem [10, 11, 14, 15, 18, 21, 25] since many years ago.

NVM (non-volatile memory) represents the storage materials with byte-addressability, persistent and high random access throughput. These materials such as PCM (phase change memory), MRAM (Magnetoresistive RAM) can provide fast random access on persistent data, improving throughout and reduce failure-recovery. Novelsm [15] and SlM-DB [14] provided some successful solutions to improve throughput and control the space amplification, there is still more space to explore.

In this work, we first evaluate the ratio of time spent on different processes during compaction. To our surprise, nearly $\frac{1}{4}$ of time was spent on converting data from file blocks to memory processable format, which is several times larger than the time spent on reading data blocks. To save the overhead of data serialization and de-serialization, this work proposes JigsawDB, a novel design that provides a united format which fits in all storage system. JigsawDB tends to solve the following problems prior proposals haven't finished: (1) **The conversion costs**: the conversion gap between stored files and in-memory data structures can be eliminated further to optimize the throughput for both read and write operations. (2) **The wear out problem**: NVM material still suffers the wear out problems, though 3d x-point materials durability is much better than NAND flash devices, directly applying random and piecemeal operations upon NVM is not that wise. (3) **To ease the impact of compaction**: in-place updates can reduce the frequency of compaction, but it is hard to accelerate this process, by redesigning the data structure on NVM, these new materials can benefit more than just use their performance advantages.

In addition to this, the benefits of organizing data in a more solid format can achieve better compression rate due to the values are logical continuous in real cases. We implement a prototype and evaluate it within an environment equipping a DRAM-emulating NVM and gain 20% speed up. We are trying to improve the system further to ease the write pause problem brought by Compaction.

2 Background

This section provides the background of NVM materials and traditional designs of LSM, introduces the basic features and usage of them, and analyzes the opportunities and challenges while combing these techniques by reviewing several prior works.

2.1 Non-Volatile Memory

NVM, or PM (persistent memory), SCM (storage class memory), is the same meaning, referring to a series of memory materials with non-volatile and byte-wise access characteristics. NVM has become a hot topic in recent years, and related research is moving forward. There are many different types of materials like PCM (phase change memory), MRAM (Magnetoresistive RAM), etc. Besides the most noticeable feature, NVM has higher throughput and storage density than traditional NAND flash devices, with the shortages like asymmetrical read/write performance, and limited lifetime leads to the wear-out problems.

Although there is no conclusion on how to use this material, research proposed several main solutions for using this material:

1. Use NVM as Persistent Transnational Memory, use methods like Redo Logging, Undo Logging, and Log-Structured to manage the transaction and data involved.

2. Use NVM as a disk to provide better random access performance on blocks and files. For example, introducing the Direct Access (DAX) in Linux. There are also file systems [13] similar to PMFS, NOVA [26], etc.

3. Combining with RDMA in a distributed scenario. Due to the high performance of NVM, the access characteristics of byte addressing, and the access mode of RDMA, distributed NVM + RDMA becomes a new architecture design.

2.2 Log-Structured-Merged Tree

LSM is designed to provide better write performance than traditional B+ trees. It has the characteristics of sequential write optimized and periodic garbage collection.

2.2.1 Sequential Write Optimized

LSM's basic idea is converting random writes to sequential writes. Most of the storage devices have the characteristics that can perform much better in sequential operations than in random access operations, no matter read or write. LSM caches the newest changes inside the memory and uses batch processing to write down those cached data. Besides, to aggregate the operations, recently updated data will overwrite in memory and be flash to the file system only for the last version.

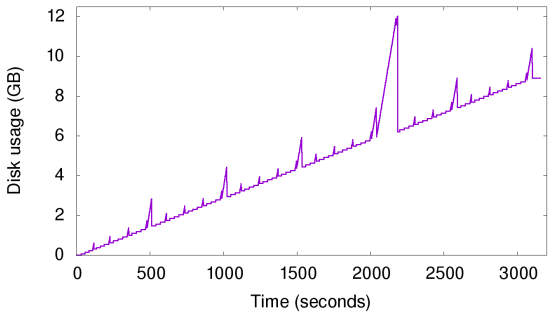


Figure 1: The obvious zigzag shape in the disk usage curve, which means the system generate a lot of useless data which is quickly discarded during the compaction process.

2.2.2 Periodic Garbage Collection

For using log-structure and an incremental update strategy to persistent operations, data stored in files may be outdated periodically due to deletions and updates, so the garbage collection process, also known as "Compaction", is needed. During this compaction process, LSM-based system will iterate through part of the persistent data, delete the out-dated data and reorganize the data in a partially sorted manner to keep read optimal.

For this *Compaction* process will walk through all key-value pairs stored in the target files, apply to merge sorting to these entries and write back to the files. Though most of the implementations tried to optimize compaction process, there are still very severe performance impact and resource occupying problems during this process. Fig. 1 shows the disk usage over time. These problems have been studied from many years ago. There are three main directions to solve these problems.

- The first one is **scheduling**, bLSM [21] proposed a "Gear Scheduler" to dispersion pressure caused by compaction. It inspired many following works, and most of the work mentioned in this paper used bLSM's results as the baseline while discussing scheduling problems.

- The second one is **algorithm optimization**, the most successful and representative one is PebblesDB in 2017 [18], use "Guard" to avoid repeated writing entries into files, easing the pressure by reducing write amplification. There are also works focus on specific workloads like LSM-trie [25]. Moreover, this may be the most popular thoughts in practice; Facebook provides several different types of compaction in RocksDB [3], [12] while Cassandra also changed many times for its compaction strategy [4].

- The third one is about **takeing advantages of new storage materials**, for new materials has been developed

lot recent years, how these new materials', like Open-Channel SSD [8] and NVM, characteristics may benefit the LSM data structure becomes an attractive topic to develop. GearDB [28] considered reorganizing the entries choosing strategy to cover the garbage collection on HM-SMR disks. FlashKV [29] use Open-Channel SSDs to optimize the compaction process's write amplification, achieved higher GC-efficiency. Novelsm [15] utilized the characteristics of NVM's byte-addressing to achieve in-place update while SLM-DB [14] combined with persistent B-Tree to organize the files into one single level and use *select-merging* to get better performance in compaction.

2.3 Opportunities and Challenges

Programming for NVM is very different from traditional memory or disk programming. This section describes several main challenges while combing the NVM with the LSM structures.

Space Amplification Space amplification problem can be the most common shortage of LSM-based systems, but NVM's high throughput of byte-addressing random operations can solve the problem to some extent. NVM's byte addressing ability allows more flexible data structure and operations; Its large capacity also benefits the buffer to store much more data, caching more operations before writing down to the sequential-based devices. From this point of view, the in-place update strategy can reduce space amplification from the very origin purpose. In addition to eliminating WAL(write-ahead-logging), Novelsm [15] applied in-place updates on NVM to reduce repeated writing and SLM-DB use persistent B-Tree to solve this problem.

Inherent Weakness of NVM Although NVM has many advantages, its inherent weaknesses cause special care in designing the data structure on NVM. Ignoring these issues may not take advantage of NVM's high throughput, and even introduce more unexpected situations or performance degradation.

Wear Out Problem: Though it is much better than traditional NAND flash devices, the limited lifetime of NVM still forces developers to consider wear out issues when designing data structures and underlying system layout [13, 22]. Introducing lifetime consideration means data structures with massively update requirements need additional operations to organize data, brings extra overheads for the system.

Asymmetrical Read/Write Performance: Another challenge brought by NVM is its asymmetrical Read and Write performance [24]. Read speed can be two to three orders of magnitude faster than write speed; this feature makes NVM more adaptable to read-intensive tasks, and need optimized batch when dealing with write-intensive tasks.

Operation	Read	Write	Encode	Decode
Ratio (extreme)	0.7%	6%	22%	5%
Ratio (average)	0.1%	5%	14%	3%

Table 1: Execution time ratio during compaction process, the second row shows a extreme case among the experiments, and the third rows shows the average value of experiment results. For different components shown in first row, **Read** means reading file blocks from disks; **Write** means writing blocks to disks; **Encode** means transfer key-value entries to the block format, mainly consists of a preamble compression for adjacent keys and variable integer encoding process etc. ; **Decode** means the process transfer blocks back into iterator's buffer and table cache.

Consistency: NVM's non-volatile feature also brings consistency problems. Data structure may result in abnormal state while flush into different storage hierarchy caused by the CPU cache and the out-of-order execution of CPU. The uncertain status of data structures means NVM requires a specified *transaction programming* model [13, 19, 23, 27] to ensure the semantics of atomic operations are achieved. Moreover, to provide consistent writing on data blocks larger than limited size (8 bytes in typical situations), logging and C-o-W (Copy on Write) is needed, which means it should consider about the alignments for data structures. Fortunately, persistent B+Tree [16, 27] was once a popular topic and has been fully developed since many years ago, providing many successful cases for us to reference.

Frequent serialization and de-serialization LSM system achieved buffering data inside memory to provide append-only writing strategy, and many of implementations use *MemTable* as an in-memory buffer and *SST* (Sorted String Table) files to store Key-Value pairs in the file system.

Table. 1. shows the time ratio of different operations, from the result we can see ***Encoding*** cost the most (almost $\frac{1}{4}$ of entire compaction process). Besides, due to the restart placeholder added by the preamble compression groups, the performance of SST files operations performed on NVM will introduce much more overhead than expect, even worse, offsets the advantages of NVM's high-performance read and write.

3 Preliminary Proposal

The challenge is to design an LSM-based system with only one single data structure which can serve both inside the memory space and persistent files. A typical unified format may suffer from low compression rate, high memory usage, and low range query speed problems, to get rid of the serialization/de-serialization overhead, the critical point is to find out a memory structure can also perform well in sequen-

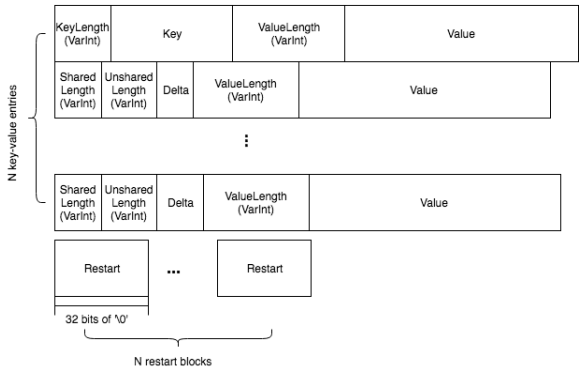


Figure 2: Original SST file block, this the format of value blocks, keys inside one same block use preamble compression to reduce space overhead.

tial writing occasions. This section analyzes several design principles and exciting results from observations.

This section introduces the abundant of applying SST files on NVM system, combining with design principles, proposes a highly aggregated, 8-byte aligned and self-indexed format adapted to all storage environments.

Abundant of SST on NVM Nearly all of the LSM systems suffer from the same shortage of disk-based DB [20]: the in-memory-abundant. Encoding/decoding, compression, value-addressing, and memory management make LSM relatively slow to apply the queries. Fig. 2 shows the original format of Value Blocks in LevelDB.

Though preamble compressed keys cost less space, the de-serialization overhead is relatively high. Moreover, in a byte-addressing memory system needs an alignment, storing an entry all together is not that efficiency, and the most important is: separated value columns become harder to take advantages of the high compression ratio feature of column-based format.

Memory Component Concern Table 2. shows some design principles of the in-memory data structure and lists out the most popular design to fit each requirement [6] (Alternative list: SkipList, HashSkipList, HashLinkList, Vector). The conclusion of this table points out the SkipList, which is the most popular data structure in typical LSM-based system, may be the best data structure to support in-memory and NVM data buffering. Moreover, Novelsm [15] and SLM-DB [14], these two successful studies have already proved that persistent SkipList can perform well on the NVM material.

KeySet Block Files Fig. 3. shows the new data structure based on SkipList and SST files. For each block, it contains two components: Key Set and Value Buffer. The **index field** is the crucial point to adapting different storing requirements. This index is a fixed length (8 bytes) filed, which can fit all

Index Efficiency	HashSkipList
Full db Scanning Cost	SkipList
Memory Overhead	Vectors
MemTable Flush Efficiency	SkipList
Concurrent Insert	SkipList
Use Case University	SkipList

Table 2: Detailed requirements and most suitable structure for *MemTable* designing

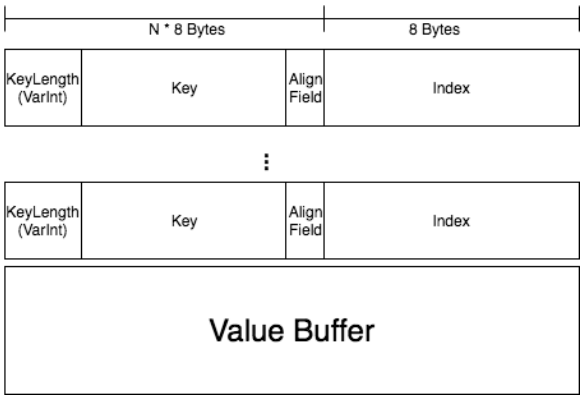


Figure 3: KeySet format, the united format works for DRAM, NVM and storage devices. For concurrency concern mentioned in section 2.3, use 8byte as the basic alignment unit for NVM. The index is a flexible field, can convert among different addressing information for all system.

addressing methods. In the DRAM, it is the pointer of its value slice; when persisted into NVM files, it can be the offset information inside target block; And for the files on disk-based storage, this field will represent a fixed 64 bits value to record an entry'sA value length.

Another component benefits both in NVM, and disk-based storage is the separated value buffer. These data blocks are raw data need no conversion. Though these data can be separated inside DRAM due to dynamic allocation mechanism, the **Flush** or (**Memtable Compaction** in another name) can aggregate these data together, become a raw and solid data. In addition, just like the column-based storage engines, gathering value data can gain better compression ratio, for the values are logically continuous.

4 Evaluation

4.1 Workload and Environment Setting

We use persistent memory workload generated from micro benchmark, perform 100000 operations of the following orders: "fillseq", "fillsync", "fillrandom", "overwrite", "readrandom", "readseq", "readrandom", "readseq", the key size is 16 bytes and value size is 100 bytes by default. The platform

Solution Cases	Disk + NVM, JigsawDB	Disk Only, JigsawDB	Disk + NVM, LevelDB	Disk Only, LevelDB
Writing Time Cost (ns)	2543407383	3100842276	2716987809	3245228467
Encoding Time Cost (ns)	2650973177	4552173256	5341669913	6872175477
Total Time Cost (s)	46	59	53	64

Table 3: Time used for different processing part, this should be the baseline for normalizations in the experiment

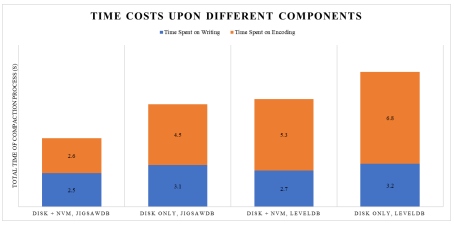


Figure 4: The absolute value of Writing and Encoding costs

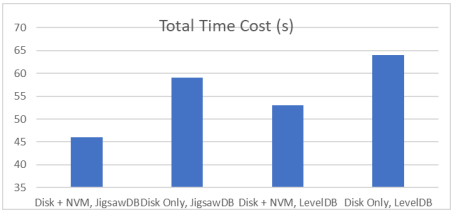


Figure 5: A more intuitive representation of overall execution time

that this experiment performed employs one CPU (8 cores, Intel i7-7700HQ, 2.80GHz) with CPUCache size of 6144 KB. It also contains 24GB DDR4 main memory, and use 8GB of DRAM to emulate the NVM as proposed in former studies [14, 15]. For the machine, Ubuntu 18.04 LTS with Linux kernel version 4.15 is used.

4.2 Result Analysis

For this is an experiment performed on a prototype, the encoding time we count here includes several data format transfer and value calculation processes rather than preamble compression only. Table. 3. shows the absolute time cost of file writing, encoding, and the entire process loop. In this experiment, we choose the ratio of writing time as the normalization indicator. For each storage system, the throughput is limited as its inherent property, the higher this value is, the system spends less overhead on conversion.

Data Normalization The sample data used in Fig. 6a. is collected among all the compaction information while performing the workload. To observe the results more intuitively, these data points are normalized into 200 equal parts according to the compaction process sequences.

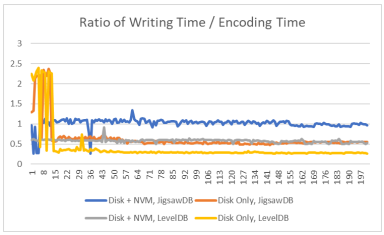
Numeric Analysis on Time Overhead From the result in Table. 3., we can conclude that JigsawDB reduces 28.1% of the overall execution time compared to the original LevelDB; and is 13.2% faster than write hierarchically on NVM. As for the effect of this united format, JigsawDB saves 22.03% of the time while hierarchy methods save only 17.19% by using NVM. The fact that saving more time by utilizing NVM proves that JigsawDB can benefits more from NVM. At last, due to the fast sequential performance of disks and limited workload size, the benefits of NVM + Disk hybrid architecture does not gain pronounced improvement, this means when data size grows big enough, the performance of JigsawDB can be even better.

Result Analysis After Normalization From the normalization result, we can find out an interesting fact: for most cases (excepts applying hybrid storage on LevelDB hierarchically) the first quarter of curves (as shown in Fig. 6b.) fluctuate significantly, but then become stabilized and remain stable for a considerable period of time(as shown in Fig. 6c. However, applying hybrid storage structures to LevelDB by extending its memory component can provide much smoother performance. The reason for this behavior is simple, at the early stage of lsm-based system, the size of a single SST file is smaller, which means there are fewer entries stored in one single SST file; in addition, with the entires being aggregated, the keys' value space is becoming dense gradually, and the complexity of finding shared data also growth.

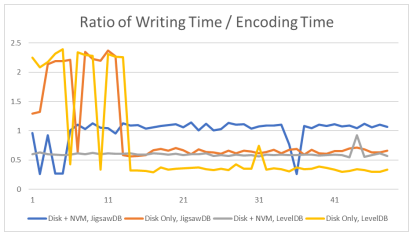
5 Discussion and Work-in-progress

In this paper we introduce the JigsawDB, a novel lsm-based system uses a united format for DRAM, NVM, and disk storage. We implement the prototype of it and evaluate this prototype to validate our design purples to reduce serialization overhead. From the experiment, Jigsaw proves this united format can save the cost of serialization and deserialization, improve the total performance against original SST files, benefit more among NVM's high throughput read and write features than naively deploy single component upon NVM. But there are still many improvements we can provide.

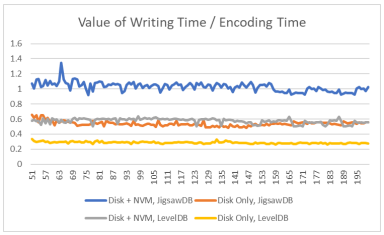
In-place Update Strategy Just as the proposal of Novelsm [15] suggests, NVM's byte addressing ability can benefit much in reducing space amplification for cutting off extra



(a) The ratio trend for the whole stage



(b) The ratio in early stage



(c) The ratio in later stage

Figure 6: These three figures shows the results of the ratio of Writing Time / Encoding Time. The ordinate in the figures is the specific proportional value while the abscissa represents the process of sampling time.

overhead of appending record produced in updating. Just as mentioned in section 3, entries can be directly addressed inside NVM, and since entries are stored in a compacted, solid data block, the traditional punch-out strategy may leave lots of sporadic memory fragments. We are working on applying a relaxation algorithm to provide in-place update and ensure won't introduce more calculating or serialization overhead.

Control the Write Pause Another extreme impact that compaction may bring is *write pause*, which is caused by sudden increment of disk usage, exhausted the bandwidth of the platform. BLSM uses the gear scheduler to reduce the impact of write pause; we are inspired by this to apply a pre-load protocol to take benefits of united format, which can distribute the pressure of the process into the NVM component to reduce the sudden increasing throughput.

References

[1] Apache cassandra. http://cassandra.apache.org/.

[2] Apache hbase – apache hbase™ home. https://hbase.apache.org/.

[3] Compaction · facebook/rocksdb wiki. https://github.com/facebook/rocksdb/wiki/Compaction.

[4] Documentation. http://cassandra.apache.org/doc/latest/operating/compaction.html?highlight=compaction.

[5] Leveldb.org. http://leveldb.org/.

[6] Memtable · facebook/rocksdb wiki. https://github.com/facebook/rocksdb/wiki/MemTable.

[7] Wiredtiger: making big data roar. http://www.wiredtiger.com/.

[8] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94. ACM, 2017.

[11] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520. ACM, 2018.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[13] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, page 15. ACM, 2014.

[14] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 191–205, 2019.

[15] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 993–1005, 2018.

[16] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386. ACM, 2016.

[17] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[18] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514. ACM, 2017.

[19] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685. IEEE, 2015.

[20] Caetano Sauer, Goetz Graefe, and Theo Härder. Fineline: log-structured transactional storage and recovery. *Proceedings of the VLDB Endowment*, 11(13):2249–2262, 2018.

[21] Russell Sears and Raghu Ramakrishnan. blsm: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 217–228. ACM, 2012.

[22] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555. ACM, 2018.

[23] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.

[24] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early evaluation of intel optane non-volatile memory with hpc i/o workloads. *arXiv preprint arXiv:1708.02199*, 2017.

[25] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 71–82, 2015.

[26] Jian Xu and Steven Swanson. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 323–338, 2016.

[27] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, 2015. USENIX Association.

[28] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on hm-smr drives with gear compaction. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 159–171, 2019.

[29] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):139, 2017.