Reduce Performance Impact of Compaction Process by Designing a Global Format for LSM

Jinghuan YU
City University of Hong Kong

Chun Jason. Xue
City University of Hong Kong

Abstract

Your abstract text goes here. Just a few facts. Whet our appetites. Not more than 200 words, if possible, and preferably closer to 150.

1 Introduction

LSM Tree is a high warm writing performance data structure proposed in 1996 [15], 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.

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 actually the same meaning, referring to a series of memory materials with non-volatile and byte-wise access characteristics. NVM has became a hot topic in recent years, and related research is moving forward. There are many different types materials like PCM (phase change memory), MRAM (Magnetoresistive RAM) etc. Beside the most obvious 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 final 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 [11] similar to PMFS, NOVA [24], 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. The most important characteristics LSM has can be concluded as 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 has 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 use batch processing to write down those cached data. In addition to simply aggregate the operations, recent updated data will overwrite in memory and be flash to the file system only for the last version.

2.2.2 Periodic Garbage Collection

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 Data stored may be outdated periodically due to deletions and updates, so garbage collection process known as "Compaction" is needed.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

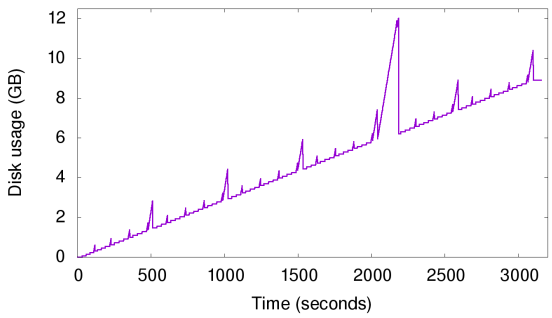

Figure 1: The significant zigzag shape in the disk usage curve, which means the system generate a lot of useless data which is quickly discarded during the compaction process.

For this *Compaction* process will walk through all key-value pairs stored in the target files, apply merge sorting to these entries and write back to the files. Though most of 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 [19] 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 base line while discussing scheduling problems.

- The second one is **algorithm optimization**, the most successful and representative one is PebblesDB in 2017 [16], use "Guard" to avoid repeated writing entries into files, easing the pressure by reducing write amplification. There are also works focus on special workloads like LSM-trie [23]. And this may be the most popular thoughts in practice, Facebook provides several different types of compaction in RocksDB [3], [10] 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 a attractive topic to develop. GearDB [26] considered reorganizing the entries choosing strategy to cover the garbage collection on HM-SMR disks. FlashKV [27] use Open-Channel SSDs to optimize the compaction process's write amplification, achieved higher GC-efficiency. Novelsm [13] utilized the characteristics of NVM's byte-addressing to achieve in-place update while SLM-DB [12] 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.

2.3.1 Space Amplification

Fundamentally, this problem is unavoidable for any log-based system as the trade off for update costs and point looking up costs. Some hardware devices like HM-SMR and Open-channel may provide raw disk control to the applications, make it is possible for application to cover this problem with device's own garbage collection process [27].

This problem can be the most typical shortage of LSM-based systems, but NVM's high throughput of byte-addressing random operations can somehow 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. This can reduce space amplification from the very origin purpose. Novelsm [13] applied in-place updates on NVM to reduce repeated writing and SLM-DB use persistent B-Tree to solve this problem.

2.3.2 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 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 [11, 20]. This means data structure with heavily update requirements like hash-table 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 [22]. 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	0.1%	3%	22%	5%

Table 1: Execution time ratio during compaction process, **read** means reading file blocks from disks; **Write** means writing blocks to disks; **Encode** means transfer key-value entries to the block format, mainly a preamble compression of adjacent keys, also include variable integer encoding process etc. ; **Decode** means the process transfer blocks back into iterator's buffer and table cache.

2.3.3 Consistency

NVM's non volatile feature also brings the consistency problem. For example, consider about the situation of double-linked table insert operation, while the following code executes to the second line and meets a sudden power failure.

```
void list_add_tail(struct cds_list_head *newp,
struct cds_list_head *head) {
head->prev->next = newp;
newp->next = head;
newp->prev = head->prev;
head->prev = newp;
}
```

For the code execution is non-volatile in the NVM, when the system is restoring after a sudden power failure, the linked list is in an abnormal state caused by the CPU cache and the out-of-order execution of CPU. This means NVM requires a specified *transaction programming* model [11, 17, 21, 25] to ensure the semantics of atomic operations are achieved, which means there is no intermediate state generated during the system is restoring, this results in the extra memory fence and cache flush operations to keep consistency. 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. Fortunately, persistent B+Tree [14, 25] was once a very popular topic, and has been greatly developed since many years ago, providing many successful cases for us to reference.

2.3.4 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 in-memory buffer and *SST* (Sorted String Table) files to store Key-Value pairs in the file system.

Use LevelDB [5] as an example, while applying a *Get* operation, LevelDB will find in the memory buffer (*Memtable*) firstly, then access the version control to check which file contains the target key-value pair; After locating the file, it still needs several iterators to transfer the file blocks to memory structure, and decode from preamble compression format to get the real value.

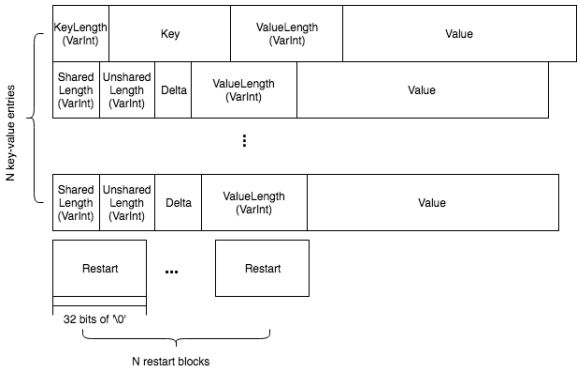

Figure 2: Original SST file block, this the format of value blocks, keys inside one same block use preamble compression to reduce space overhead.

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). In addition, 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 a LSM-based system with only one single data structure which can serve both inside the memory space and persistent files. Typical united 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 key point is to find out a memory structure can also perform well in sequential writing situation. This section analyzes several design principles and interesting results from observations.

This section introduces the abundant of simply apply 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.

3.1 Abundant of SST on NVM

Nearly all of the LSM systems suffer from the common shortage of disk-based DB [18]: the in-memory-abundant. Encoding/decoding, compression, value-addressing and memory management make LSM is relatively slow to apply the queries. Fig. 2 shows the original format of Value Blocks in LevelDB.

Though preamble compressed keys can store in less space, the de-serialization overhead is relatively high. Moreover, in a byte-addressing memory system needs alignment, storing

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

an entry all together is not that efficiency, and the most important is this get really hard to take advantages of the high compression ratio feature of column-based format.

3.2 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 [13] and SLM-DB [12], these two successful studies has already proved that persistent SkipList can perform well on the NVM material.

3.3 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 key point to adapting different storing requirements. This index is a fixed length (8 bytes) filed, which can fit all 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 filed will represent a fixed 64 bits value to record an entry's value length.

Another component can benefit 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 other 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

workload setting and environment configuration We use persistent memory workload generate from micro benchmark, provide random filling, sequential filling tests. The platform that collected the results employs one CPU (Intel i7-7700, 3.4GHz), and 24GB main memory, 8GB among this memory is used to emulate the NVM as in former studies [12, 13] . For the machine, Ubuntu 18.04 LTS with Linux kernel version 4.15 is used.

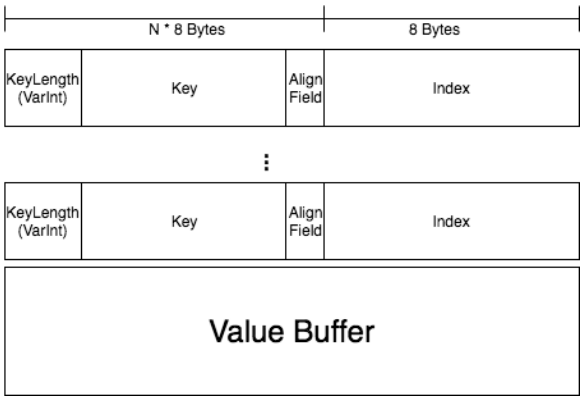

Figure 3: KeySet format, the united format works for DRAM, NVM and storage devices. For concurrency concern mentioned in section 2.3.3, use 8byte as the basic alignment unit for NVM. The index is a flexible field, can convert among different addressing information for all system.

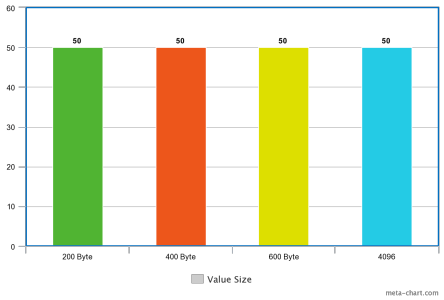

Figure 4: The introduce of space amplification, the different reasons' ratio in entire result, point out most of the amplification problem is caused by the Compaction

Result Analysis Here goes the result analysis

5 Conclusion and Work-in-progress

From this prototype of implementation we provide a united format for DRAM, NVM and disk storage, this united format can save the cost of serialization and de-serialization, improve the total performance against original SST files, benefit more among NVM's high throughput read and write features. But there are still many improvements we are working on.

In-place Update Strategy Just as the proposal of Novelsm [13] suggests, NVM's byte addressing ability can benefit much in reducing space amplification for cutting off extra overhead of appending record produced in updating. Just as mentioned in section 3.3, records can be directly addressed inside NVM, and since entires are stored in a compacted, solid data block, traditional punch-out strategy may leave lots of sporadic memory fragments. We are working on applying 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, and inspired by this, we are working on 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/. (Accessed on 03/07/2019).

[2] Apache hbase – apache hbase™ home. https://hbase.apache.org/. (Accessed on 03/07/2019).

[3] Compaction · facebook/rocksdb wiki. https://github.com/facebook/rocksdb/wiki/Compaction. (Accessed on 03/10/2019).

[4] Documentation. http://cassandra.apache.org/doc/latest/operating/compaction.html?highlight=compaction. (Accessed on 03/10/2019).

[5] Leveldb.org. http://leveldb.org/. (Accessed on 03/07/2019).

[6] Memtable · facebook/rocksdb wiki. https://github.com/facebook/rocksdb/wiki/MemTable. (Accessed on 03/11/2019).

[7] Wiredtiger: making big data roar. http://www.wiredtiger.com/. (Accessed on 03/07/2019).

[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] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.