2023 IEEE 39th International Conference on Data Engineering (ICDE) | 979-8-3503-2227-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDE55515.2023.00258

PM-Blade: A Persistent Memory Augmented LSM-tree Storage for Database

Yinan Zhang
East China Normal University[†]
Shanghai, China
ynzhang@stu.ecnu.edu.cn

Huiqi Hu[*]
East China Normal University[†]
Shanghai, China
hqhu@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University[†]
Shanghai, China
xzhou@dase.ecnu.edu.cn

Enlong Xie
Meituan
Beijing, China
xieenlong@meituan.com

Hongdi Ren
Meituan
Shanghai, China
renhongdi02@meituan.com

Le Jin
Meituan
Beijing, China
jinle02@meituan.com

***Abstract*—In this paper, we present *PM-Blade*, an LSM-tree structured storage augmented with persistent memory (or non-volatile memory). *PM-Blade* utilizes persistent memory to optimize read performance and reduce write amplification, which are essential to Meituan's online retail applications. Distinguished from existing designs, *PM-Blade* leverages persistent memory to drastically increase the capacity of the level-0 layer of LSM-tree. An enlarged level-0 layer allows a large amount of hot or warm data to be retained in persistent memory, enabling high read performance. At the same time, it works as a large write buffer that absorbs write amplification. To make the best of the design, we devised an internal compaction method and used a cost-based compaction strategy to maximize the utility of the level-0 layer. We implemented the compaction method using coroutines to improve its efficiency and resource utilization. We evaluated *PM-Blade* through extensive experiments, in which *PM-Blade* outperformed several open-source alternatives on standard benchmarks and a real-world workload of Meituan.**

***Index Terms*—LSM-tree, Persistent memory, Database storage**

I. Introduction

Meituan is a leading e-commerce service provider in China, serving hundreds of millions of online users. During peak hours, Meituan's systems receive tens of millions of orders per hour. Such a huge transaction processing pressure prompted Meituan to consider developing its own OLTP database system, to replace traditional databases (e.g., MySQL). Meituan's database group has developed a distributed database system named *Blade* database, which has been deployed in several online retail applications, such as those for take-out ordering, intra-city delivery, lodging and advertising.

Blade's storage needs to handle huge write-intensive workloads generated by online retail ordering. Log-Structured Merge-tree (LSM-tree), which is considered a write-friendly storage structure, has been widely adopted by modern database systems [1]–[3]. It thus fits Meituan's scenario well. LSM-tree first buffers all writes in memory and subsequently compacts them to SSDs through sequential I/Os. This design improves write performance by eliminating random writes, at the cost of degraded read performance and considerable write amplification [4]. Unfortunately, neither read performance degradation nor write amplification can be ignored in online retail applications. Further optimization seems indispensable.

An important aspect is the temporal locality of online retail workloads. There is usually a clear division between hot and cold data. For instance, consider the lifecycle of a take-out order. An order will undergo many updates, regarding its payment status, packing status and delivery status, after being placed (i.e., considered hot data). Once it is finished, it will become a part of the recent history and be frequently queried (i.e., considered as warm data). Over time, it gets cold and is rarely accessed. This apparent temporal locality affects both read and write amplification for the LSM-tree. First, it results in frequent random reads on hot and warm data (updates will also come with reads), which requires high performance and rapid response of reads. Second, hot data will be frequently updated, which may cause severe write amplification. Moreover, secondary indexes, implemented as index tables in LSM-trees, will often be employed to speed up the lookup. These index tables, although small in size, will be updated randomly, which will result in severe write amplification too. Therefore, it requires special techniques for efficiently merging updates into SSTables to curb the write amplification of the LSM-tree.

A direct solution to optimizing the access to hot and warm data is to fit them in the memory. However, the expansion of DRAM is expensive. The upper limit of DRAM capacity of today's computers is still low, making holding all the hot and warm data in DRAM unrealistic. To address the problem, we consider using Persistent Memory (PM), Intel Optane in particular, to expand the memory used by LSM-tree. PM can provide permanent data storage with a high memory density. Meanwhile, it outperforms SSD significantly in access latency. A LSM-tree adopts a multi-level structure, where the top level is called level-0, which contains a significant proportion of hot

[*] is the corresponding author.
[†] Shanghai Engineering Research Center of Big Data Management.

and warm data. Traditional LSM-trees deploy level-0 on SSD, resulting in poor performance on random access to hot and warm data. In contrast, we consider deploying level-0 on PM, expecting it to boost the performance of LSM-tree in online retailing.

Early efforts have revealed that optimizing the LSM-tree with PM is promising [5], [6]. Databases such as PolarDB (X-engine [2]) and MariaDB [7] have used PM to successfully improve the performance of their storage layers. Some recent works have used PM as level-0 to minimize write stall and write amplification [8], [9]. Nevertheless, we found that the potential of PM at level-0 has not been fully exploited. At least, the following two directions can be further considered for its optimization.

Opportunity 1: Utilizing the large capacity of PM. Compared with DRAM, the capacity of PM is much larger, but the previous work did not make good use of this. For instance, MatrixKV [9] only allocates 8GB of PM to level-0. Considering that the capacity of a PM device is usually 128GB or 256GB, a significant amount of its space is underutilized. Compaction from level-0 to level-1 often incurs severe write amplification and write stalls. Using a large level-0 can alleviate this problem [9]. With a larger level-0, the frequency of compaction can be reduced, which can further relieve the pressure on the SSD. A side effect of a large level-0 is its negative impact on read performance. When memory tables are directly flushed down to level-0, they are not ordered. Thus, accessing the particular data requires scanning through the whole level-0. While it is feasible to build some indexes in level-0 [8], [9], it would result in high maintenance and space costs. As data in level-0 will soon be compacted to the next layer, the benefit of indexing is also limited – data may not be accessed before being swapped out.

Opportunity 2: Fully utilizing the read performance of PM. PM is only slightly slower than DRAM in read latency (about 3-5 times slower [10]). We implemented an array-based structure on PM that supports binary search and inserted 1 million data entries into it. We compared its performance against two versions of LSM-trees. One version had all the data in SSD. The other version had all the data in the DRAM cache. Table I shows the result. This shows that the read performance of the table on PM is close to that of hitting the cache. This high speed of PM provides much potential to improve the read performance of the LSM-tree storage. This indicates that many hot or warm data items can be placed on the PM to provide a fast read, particularly given its large capacity noted above.

TABLE I
COMPARATION OF QUERY LATENCY

The number of tables	1	2	4	8
Table on PM	3.3 us	4.4 us	7.9 us	14.5 us
SSTable in cache	2.6 us	3.5 us	6.0 us	10.7 us
SSTable in SSD	22.3 us	31.3 us	49.9 us	100.2 us

PM also provides high read and write bandwidth. Compaction in LSM-tree is an expensive operation that consumes I/O bandwidth [4]. Leveraging the high bandwidth of PM, if we can shift some compaction workload to PM, the I/O cost on SSD can be reduced. This provides more design space for compaction.

Based on the above insights, we implement *PM-Blade*, which works as a storage component of Meituan's Blade database. The core design philosophy of *PM-Blade* is to use the PM as a high-capacity level-0 layer. This PM layer serves two purposes. First, it provides high-performance reads for hot and warm data. As mentioned above, the read latency of PM is close to that of the DRAM cache. By keeping as much hot and warm data as possible in PM, we can significantly improve the read performance. Second, it absorbs write amplification and reduces I/O overhead on the SSD. In *PM-Blade*, the PM-based level-0 functions as a large write buffer, which can absorb the amount of flushing traffic from DRAM and relieve the pressure on SSD. With a large level-0, *PM-Blade* does not compact data from level-0 to level-1 frequently. Instead, a significant number of compaction can be finished in PM, which can reduce write amplification caused by frequent updates.

A number of challenges need to be addressed for *PM-Blade* to work. (i) As mentioned earlier, data tables in level-0, which we call *PM tables*, are unordered. As the size of level-0 increases, this hurts read performance. Our solution is to perform internal compaction within the PM-based level-0 to convert unsorted PM tables to sorted ones. (ii) To achieve an overall performance improvement, compaction within level-0 needs to be managed. To this end, we partition the LSM-tree and perform cost-based compaction in each partition. (iii) To make better use of the PM space, we perform compression on PM tables. (iv) Using PM as level-0, compaction from PM to SSD becomes critical to the performance. Thus, we introduce a parallel compaction mechanism to maximize the usage of I/O and CPU resources of major compaction.

The contributions of this paper can be summarized as follows:

- We designed *PM-Blade*, which employs a high-capacity PM-based level-0 layer to optimize the LSM-tree in online retail applications. It applies holistic optimization, which considers read performance, write amplification as well as compaction efficiency.
- We proposed a number of techniques to optimize the PM-based level-0 layer. A compressed structure is used to economize PM space. An internal compaction method and a cost-based compaction strategy are proposed to boost its read efficiency, reduce its write amplification and improve its hit ratio.
- We observed unnecessary CPU waiting time and bursty I/O contentions during compaction. We devised a coroutine-based compaction method, which performs precise scheduling to reduce CPU waiting time and I/O contention.
- We have evaluated *PM-Blade* through comprehensive experiments and compared it with several open-source systems. The results demonstrate the superiority of *PM-Blade* over the alternatives.

The rest of this paper is organized as follows. Section II describes the related work. Section III presents the overall design of *PM-Blade*. Section IV gives the detailed design of the PM-based level-0. Section V introduces the coroutine-based compaction method. We evaluate our system in Section VI and conclude the paper in Section VII.

II. Related work

A. Optimization of LSM-tree

LSM-tree has been studied intensively. Chen et al. [4] provided a taxonomy of its improvements. It showed that an LSM-tree can be optimized from many perspectives. When LSM-tree is used as database storage, write amplification, read performance and compaction performance become the major concerns. So, we focus on these three aspects in this paper.

Write amplification. To deal with write amplification, three strategies are commonly employed: tiering, key&value separation, hot&cold separation. The core idea of tiering [11]–[15] is to further relax the LSM-tree's data ordering. Multiple SSTables at each level are compacted only once before they are pushed to the next level [4]. However, tiering requires reading more SSTables to find the target key, which adversely affects read performance.

The core idea of key&value separation [16]–[18] is to store the value in a separate data structure, such as a value log. In this way, only the keys are compacted during compaction, which reduces write amplification. However, this method requires additional overhead to maintain value logs. As the order of values is no longer consistent with that of keys, it is unfriendly to scan operations. The strategy seems ineffective in our scenario, in which scan operations might not be avoidable.

Hot&cold separation [14], [19] focuses on the impact of frequent updates on write amplification. The previous work found that a small amount of frequently updated keys can be associated with a large number of old SSTables, resulting in severe write amplification. Therefore, the core idea of this method is to store the frequently updated keys in a separate data structure to reduce the number of SSTables involved in compaction. However, this method needs to maintain an additional persistent data structure, such as a hot data log. We use PM to accommodate hot data in this paper, as an orthogonal approach.

Read performance. Many works attempted to improve the read performance of LSM-tree using Bloom filter and Hashing. Bloom filter [20], [21] is the most common method to improve the read performance. The original LSM-tree maintains one or more Bloom filters for each SSTable. A read operation can first assess whether the target key exists in the current SSTable using the bloom filter. If the key does not exist, the SSTable can be skipped to reduce invalid seek overhead. To obtain the best performance of point read, some works adopt hashing. For example, LSM-trie [22] partitions SSTables through hashing. When performing point read, the system can quickly locate the potential partitions for the keys. ChameleonDB [23] redesigned SSTables as hash tables, which are more friendly to point read.

Bloom filter and hashing only benefit point read. They are not enough for general database workloads, which may contain a large number of range queries.

Compaction Optimization. Compaction usually consumes a lot of I/O resources, resulting in periodic latency spikes. Therefore, Silk [24], [25] dynamically allocates I/O bandwidth to compaction according to the running workload, to prevent latency spikes. A compaction process can be separated into three stages: read, sort, and write stages, that can run in pipelines. Therefore, Zhang et al. [26] proposed pipeline-based compaction to improve the CPU and I/O parallelism. We further improve the compaction resource utilization with a coroutine-based method in this paper.

B. LSM-tree Storage Augmented with Persistent Memory

A number of works have explored how to use PM to optimize LSM-trees. They tried using PM as different components of an LSM-tree, which include (i) the memtable, (ii) the level-0 and (iii) an auxiliary structure.

Augmenting memtable with PM. The memtable of LSM-tree can benefit from PM in two ways. First, we can use PM to reduce the recovery time of memtable. NoveLSM [5] uses PM to expand the capacity of memtable. As the contents in PM are persistent, they speed up database recovery. SLM-DB [6], in contrast, redesigned the entire memtable on PM to completely eliminate the recovery overhead. Second, we can enlarge the memtable using a DRAM-PM hybrid memory. Yan et al. [8] and ListDB [27] designed a semi-persistent memtable. They store data in PM while maintaining the indexes in DRAM, which can make use of both the high performance of DRAM and the large capacity of PM. However, using PM to augment memtable does not help reduce write amplification and write stall [9]. In addition, it may reduce the performance of the memtable, which is critical for the overall performance of the LSM-tree.

Augmenting level-0 with PM. Using PM as level-0 brings additional benefits. In particular, it solves the write stall problem. Previous work has found that the compaction from level-0 to level-1 will cause the most write amplification and write stall [9]. Therefore, MatrixKV redesigned level-0 using PM and proposed a column compaction technique, which performs fine-grained compaction from level-0 to level-1 to avoid write stalls. To optimize the read performance of the PM-based level-0, MatrixKV proposed a cross-hint search method. Yan et al. [8] further built indexes in DRAM to improve the read performance.

However, these two approaches did not consider the large capacity of PM. They only allocate small-sized PM to level-0s. In this paper, we show that a large-capacity PM-based level-0 can remarkably improve the performance of LSM-tree, although it requires extra techniques to facilitate the level-0.

Hosting auxiliary structures on PM. PM's great performance and byte-addressability make it ideal for hosting auxiliary structures that can speed up LSM-tree reads. SLM-DB [6] designed a global index for all SSTables on the PM, which stores all keys in the LSM-tree. However, maintaining

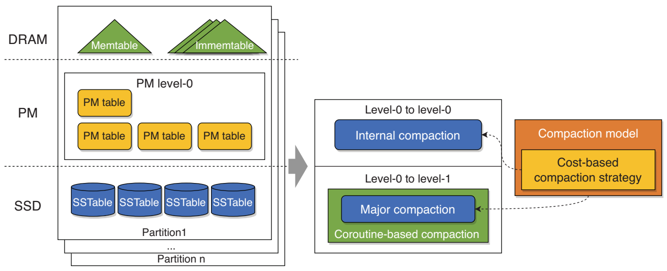


Fig. 1. Architecture of *PM-Blade*

such a global index incurs extra costs. We do not discuss this approach further in this paper, as it is orthogonal to ours.

III. Architecture of PM-Blade

The main components of *PM-Blade* are shown in Figure 1. *PM-Blade* adopts a three-tiered structure. The top tier is the memtable in DRAM. The second tier, level-0, is constructed using PM, while the third tier, level-1, is built on SSD. Conventional LSM-trees use multi-tiered structures, which have several disadvantages. First, it may incur more write amplification as the number of tiers grows [4]. Second, too many tiers hurt read performance (especially to scan operations), because each read operation has to traverse multiple layers.

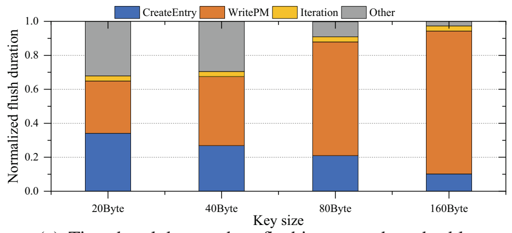
Partitioned LSM-trees. A negative effect of using one layer of SSTables comes from too many intersections between level-0 and level-1, which makes the compaction from level-0 to level-1 expensive. Overly intensive compaction can also cause a write stall, i.e., blocking other data access operations. To address this problem, *PM-Blade* divides the data (as well as the LSM-tree structure) by range into partitions, such that each partition is managed independently. Different partitions will usually observe different access features, e.g., different read/write ratios and update/insert ratios. Thus, the workload of compaction is distributed more evenly over different times.

Level-0 in Persistent Memory. To make the most of PM's high capacity, we redesigned the structure and the function of level-0, which is composed of a large number of *PM tables*. A memtable is flushed from DRAM to level-0 as a PM table. To save space and access costs, each PM table uses a compressed structure to store its data. Level-0 performs compaction within itself, which is called internal compaction. Internal compaction sorts PM tables in level-0 and deletes redundant data. More importantly, to take full advantage of the high performance of PM, it adopts a cost-based compaction strategy, which allows it to identify the optimal compaction path. See Section IV for more details.

Coroutine-based Compaction. Compaction from level-0 to level-1 is called major compaction. As major compaction consumes I/Os, its performance is crucial to the whole system. We observed unnecessary CPU waiting time and bursty I/O contentions during major compaction. Therefore, *PM-Blade* utilizes coroutine to parallelize the compaction process. Details are further introduced in Section V.

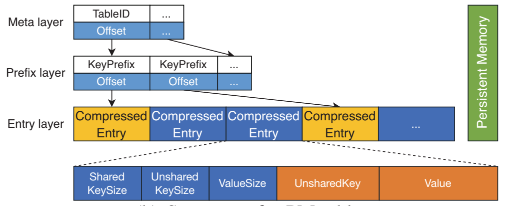
IV. Details of Level-0 with PM

This section presents the detailed design of the PM-based level-0 in *PM-Blade*.



(a) Time breakdown when flushing array-based table

Encoding	Key	Value
Record table	{tableID}_{rowID}	[col1,col2,...]
Index table	{tableID}_{indexedColValue}_{rowID}	null



(b) Structure of a PM table

Fig. 2. Data compression on PM tables

A. Compressing PM tables

It is straightforward to use an array-based table on PM, which consists of a data array and a metadata array [9]. The data array contains sorted key-value pairs. To support variable length values, the metadata records the offset of each key in the data array. We first created an array-based table on PM and performed minor compaction with it. Figure 2(a) shows the time breakdown of flushing it to level-0. As we can see, more than half of the time is spent on writing to persistent memory devices when the data size is greater than 40B. Compression would allow us to achieve better flushing performance. Therefore, *PM-Blade* applies lightweight compression to the structure of the PM table. The benefit of compression is two-fold. First, it saves the space of PM. Second, it enables better performance in both write and read.

Choosing a right compression algorithm is critical to the read performance of level-0. Common compression techniques, such as Snappy [28] and LZ77 [29], may incur substantial decompression costs, which may hurt read performance. Lightweight compression approaches, such as prefix compression, can be more cost-effective. Figure 2(b) shows the encoding of a database table. Most keys share common prefixes such as {tableID}, particularly on indexed tables. This indicates that prefix compression can achieve a good compression ratio too. Hence, *PM-Blade* applies prefix-based compression to level-0.

Figure 2(b) shows the structure of a PM table. A PM table uses a three-layer structure: (i) a meta layer – because a PM table can contain entries from multiple database tables, we extract the superfluous coding information, such as {tableID}s, and save them in the meta layer; (ii) a prefix layer – we further partition keys into groups, each consisting of eight or sixteen elements, extract a fixed-length prefix for each group's first keys, and store it in the prefix layer; (iii) an entry layer – key-value pairs are organized into entries (with prefixes removed) and stored in this layer.

This three-layer structure helps improve read performance. If we use a simple array-based structure to perform a binary search for a data entry, we need to access the PM twice for each lookup, once for the metadata and once for the data. In the three-layer structure, we first perform a binary search on the prefix layer to locate the group containing the key. Then, we conduct a sequential search in the group to find the requested entry. As the prefixes are fixed-sized, a binary search on them will be efficient. A sequential scan of an entry group also incurs low costs on PM.

B. Internal Compaction

In a traditional LSM-tree, when level-0 is almost full, the major compaction will be triggered. Such a simple design leaves two performance issues. First, when level-0 is close to full, its read performance will be bad, as each read needs to scan through a large number of unordered PM tables. Second, after the compaction, level-0 will be emptied, leaving the precious PM space underutilized. Hence, we opt to perform internal compaction within level-0.

Figure 3 shows the process of an internal compaction in an LSM-tree partition. When a minor compaction occurs, an immutable table will be persisted to level-0 as a PM table. These PM tables are unordered with respect to each other and called unsorted tables. Unsorted tables cause *read amplification* (a notation in [6]), as all of them need to be accessed during a read. Once level-0 has accumulated a certain number of unsorted PM tables, *PM-Blade* invokes an internal compaction, which will pick all unsorted PM tables and compact them into the sorted PM tables. After the internal compaction, all PM tables will become sorted with respect to each other. Then, level-0 will start accumulating unsorted tables again, until the subsequent internal compaction is triggered.

Internal compaction improves the read performance. At the same time, it can also reduce write amplification on the SSD and improve the space utilization of PM by removing redundant data. Moreover, it reduces the frequency of major compaction by enabling level-0 to accommodate more data through merging the PM tables. Since internal compaction is performed on PM, its overheads are manageable.

C. Compaction Models

The compaction models guide the scheduling of compaction. Our compaction models have the following three objectives:

- alleviate the problem of read amplification at level-0 as the number of PM tables grows;
- make full use of PM space and reduce write amplification on SSDs;
- keep warm data in PM to reduce the I/O cost of read.

Cost-based Compaction. We recognize that the three objectives do not necessitate the same timing for the compaction. First, if PM has unlimited space, we should put all the data on PM because PM is always faster than SSDs. In other words, we only need to trigger a major compaction when the PM is about to run out of space. Likewise, if the PM has sufficient space,

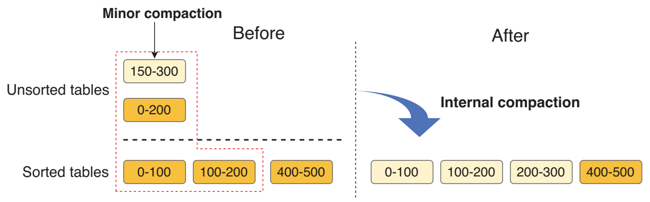


Fig. 3. Internal compaction

TABLE II
NOTATIONS IN COMPACTION MODELS

Notation	Definition
p_i	a partition on PM denoted via subscript i
s_i	space size of partition p_i
τ_m	space threshold to trigger major compaction
τ_w	threshold to trigger internal compaction for write amplification
τ_t	space threshold for selecting partitions to be preserved in PM
n_i	number of unsorted PM tables for partition p_i
n_i^r	number of reads for partition p_i
n_i^w	number of writes for partition p_i
n_i^u	number of updates for partition p_i
$\hat{n_i^r}$	number of reads for partition p_i per second
$\mathcal{I}_p$	cost for inter compat a record
$\mathcal{I}_s$	cost for major compat a record
$\mathcal{I}_b$	cost for binary search a record on a PM table
$\hat{t_p}$	time for internal compact a record
Φ	set of partitions selected to be preserved in PM
P	set of all partitions

write amplification on the SSD cannot be reduced by internal compaction but itself produces additional cost. Finally, only reducing read amplification provides a performance benefit by converting unordered tables to ordered tables via internal compaction even when PM space is sufficient. As a result, we consider compaction at different times to meet each of the three objectives.

We propose three cost models (which will be presented subsequently) to guide the decision-making. For ease of description, we list all the symbols used in the cost models in Table II. Algorithm 1 gives the procedure of compaction strategy, When a new PM table has been flushed to a partition(p_i, it first evaluates Equation 1 to decide whether to invoke internal compaction(lines 1-2) to speed up read. Then, it tests if the capacity of the PM table in p_i has exceeded a space threshold τ_w (using Equation 2 (lines 4-5)), and decides whether to invoke internal compaction to reduce the write amplification on SSD. Finally, it determines whether the capacity of the whole level-0 has exceeded a limit(τ_m) (through Equation 3 (lines 7-8)) and selects appropriate partition set to perform major compaction.

Alleviating Read Amplification. While internal compaction can benefit read performance, performing it at an inappropriate time will cause unnecessary overhead. Intuitively, when a partition contains only a small number of unsorted tables or is not frequently read, internal compaction is not needed. Internal compaction will be necessary when the number of unsorted tables grows to a certain threshold.

Whenever a new unsorted table is formed for a partition p_i, we examine whether we need to perform internal compaction on this partition. Suppose that there are n_i unsorted tables and m_i sorted tables in the level-0 of p_i. Then, for each read,

Algorithm 1: Compaction Strategy

```
  // When a PM table has been flushed to $p_i$
1 if $\Delta\widehat{cost_i(rf)} > 0$ then
      // according eqution 1
2     conduct internal compaction for $p_i$
3 end
4 if $s_i \geq \tau_w$ && $\Delta cost_i(wf) > 0$ then
      // according eqution 2
5     conduct internal compaction for $p_i$
6 end
7 if $s_o \geq \tau_m$ then
      // according eqution 3
8     select partition set $P - \Phi$ for major compaction
9 end
```

at most n_i+1 PM tables need to be looked up before the internal compaction, and after the compaction, only one PM table needs to be looked up. Each PM table lookup is a binary search, whose cost is denoted as $\mathcal{I}_b$. Therefore, the expected cost benefit for a read after internal compaction is $\frac{n_i}{2} \cdot \mathcal{I}_b$. Conversely, the cost incurred by internal compaction is the merge sorting of n_i+m_i PM tables. We assume that each PM table has N data. Then, the average cost of internal compaction to process a record is $\mathcal{I}_p$. The total cost consumption of internal compaction is $(n_i + m_i) \cdot N \cdot \mathcal{I}_p$. Since the time at which an internal compaction yields benefits is indeterminable, we compare the rates of the two costs, i.e., the cost benefit per unit of time. Finally, the cost benefit rate of p_i can be calculated as follows:

$$\begin{aligned}\Delta\widehat{cost_i(rf)} &= \hat{n_i^r} \cdot (\frac{n_i}{2} + 1) \cdot \mathcal{I}_b - \hat{n_i^r} \cdot \mathcal{I}_b - \frac{(n_i + m_i) \cdot N \cdot \mathcal{I}_p}{(n_i + m_i) \cdot N \cdot \hat{t_p}} \\ &= \hat{n_i^r} \cdot \frac{n_i}{2} \cdot \mathcal{I}_b - \frac{\mathcal{I}_p}{\hat{t_p}}\end{aligned} \tag{1}$$

where $\hat{n_i^r}$ is the number of reads on p_i per second, $\hat{t_p}$ is the average time it takes for internal compaction to process a record. When $\Delta\widehat{cost_i(rf)} > 0$, we need to trigger the internal compaction for partition p_i.

Reducing write amplification on SSD. Internal compaction can also be used to minimize redundancy among PM tables, so as to reduce the frequency of major compaction. This in turn reduces write amplification on SSD. This is only effective when there is enough duplicate data among PM tables. Therefore, the cost benefit can be calculated as:

$$\begin{aligned}&\Delta cost_i(wf) = (n_{bef} - n_{aft}) \cdot \mathcal{I}_s - n_{bef} \cdot \mathcal{I}_p \\ &\quad wrt. \quad n_{bef} \approx n_i^w, \\ &\quad\quad\quad\quad n_{aft} \approx n_i^u.\end{aligned} \tag{2}$$

where n_{bef} is the number of record before internal compaction, n_{aft} is the one after internal compaction. Therefore, $(n_{bef} - n_{aft}) \cdot \mathcal{I}_s$ is the saved cost on SSD in the subsequent major compaction, where $\mathcal{I}_s$ is the average cost of major compaction to process a record. $n_{bef} \cdot \mathcal{I}_p$ is the cost on PM in the internal compaction. It is difficult for us to actually know what the number of data records is after eliminating redundancy (i.e., n_{aft}). Since duplicate data in PM tables are generated by updates in LSM-tree, we use the number of updates(n_i^u) on the partition (p_i) to estimate it. Likewise, we use the number of writes on p_i to estimate n_{bef}.

Keeping Warm Data in PM. It is unnecessary to migrate all data from PM to SSD during a major compaction. Instead, if we retain the warm data in PM, it can benefit subsequent data access. Therefore, we need to decide which partition's data to be preserved in PM. This can be determined by the hotness of data, namely, how frequently will be read. We calculate the hotness of a partition by the frequency of its recent read accesses. Let n_i^r denote the number of reads on the partition p_i. The larger the value of n_i^r, the hotter the data. Then, we can formalize the problem as a backpack problem, which requires choosing the subset Φ of the hottest partitions, whose capacity is less than or equal to τ_t:

$$\begin{aligned}&max \sum_{p_i \in \Phi} n_i^r \\ wrt. &\sum_{p_i \in \Phi} s_i \leq \tau_t\end{aligned} \tag{3}$$

where s_i is the size of partition p_i. Since the complexity of solving the backpack problem is NP-complete [30], we apply a greedy strategy that selects the partitions with the maximum $\frac{n_i^r}{s_i}$ until the overall size reaches τ_t. Moreover, we can adjust τ_t. When the system is mainly serving reads, the data accumulation on PM will be slow. Then we can increase τ_t, to leave more data in PM.

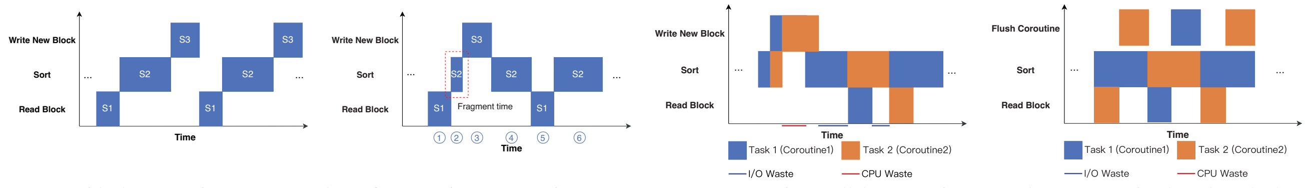
Setting Parameters. In detail, we should set up the parameters used. s_i are continuously observed and recorded. n_i^r, n_i^w and n_i^u are updated periodically to reflect their recent status. They will be re-zeroed when a major compaction or internal compaction occurs. $\mathcal{I}_p$,$\mathcal{I}_s$ and $\mathcal{I}_b$ are tunable scalars that can be set according to devices performance.

V. Coroutine-based Compaction

In this section, we present a coroutine-based approach to parallelizing major compaction. First, we show that the operating system threading is ineffective in scheduling parallel compaction, as it cannot utilize the CPU and I/O resources well. In particular, the erratic behaviors in compaction causes unnecessary CPU waiting time. Then, we introduce a method that uses coroutines to improve CPU and I/O utilization.

A. Thread scheduling is not conducive to compaction

Task scheduling plays an important role in achieving good performance. However, scheduling tasks by the operating system does not result in optimal resource utilization. We conducted a test that ran multiple compaction tasks (one thread per task) on a single core and observed the CPU and I/O resource usage. As the results in Table III show, the CPU and I/O devices are quite idle throughout the compaction



(a) An ideal compaction process (b) A fragment in a compaction process (c) CPU waste in parallel compaction (d) Our coroutine-based method

Fig. 4. Behaviours of different compaction processes

TABLE III
RESOURCE UTILIZATION OF COMPACTION WITH MULTI-THREADS

The number of threads	1	2	3	4	5
Time speed up	1x	1.6x	1.8x	1.9x	1.9x
CPU idleness	43.1%	34.5%	32.8%	31.1%	30.1%
I/O device idleness	47.1%	39.8%	38.7%	37.4%	36.7%
I/O latency	3.9ms	4.7ms	6.4ms	8.4ms	10.9ms

process. Increasing the number of threads does not help in improving their performance. This is because the operating system strives to maximize fairness and cares less about CPU and I/O utilization when scheduling threads. If improperly scheduled, compaction threads, which alternate between CPU-intensive and I/O-intensive operations, may all be stuck in I/O contention at some point.

We utilize coroutines to achieve more precise scheduling and better resource utilization. Our approach aims to reducing CPU waiting time and bursty I/O contention at the same time. In theory, I/Os would not block CPU threads during parallel compaction. In reality, it may occur when there are erratic behaviors in the course of compaction. Next, we further describe the details.

B. Erratic behaviors in compaction process

We discovered that the compaction process in practice differs from what is commonly recognized. Figure4(a) was typically used to illustrate the compaction process [26]. In a nutshell, the compaction process can be broken down into three stages: read block(S1), sort(S2), and write new block(S3), S1 and S3 are I/O operations. Note that in the past, S1 and S3 dominated the compaction because data had to be read from and subsequently written to the SSD. After using PM as level-0, there are more memory operations, which makes S2 last longer. Ideally, a process cycle through S1, S2, and S3 with each loop having a similar duration for S1, S2, and S3. However, in practice, we discover that the process displays erratic behaviors, where each S2's execution time or start timing for S3 is erratic. As an example shown in figure4(b), a stage of S2 is divided into smaller time clips by S3, where one S2 has a much shorter time than the others.

To explain why Figure 4(b) actually turns out to be the case, we use an example in Figure 5 to illustrate it. In practice, the compaction process places the data to be written by S3 in a write buffer and the data to be read by S1 in a read buffer. These buffers have limited capacity. When the write buffer reaches its capacity, it is flushed to disk immediately. In Figure 5, we can observe that after S2 completes its execution (step ②), fills the write buffer, and then start S3 immediately (step ③). As we can also observe from the procedure, S2 will

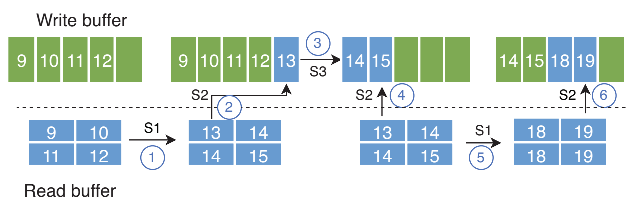


Fig. 5. An example of the compaction process

discard any duplicated data(data 14 of step ④, data 16 and 17 of step ⑥), it is the workload that affects how quickly the write buffer fills. As a result, the timing of triggering each S3 becomes unpredictable. We call those S2, that are considerably shorter than others *fragments*.

As a result of the difficulty in predicting the execution patterns of the three phases, the straightforward method to make the compaction process parallel is inefficient. We identify that the existence of fragments is the main reason that causes unnecessary CPU idleness. Figure4(c) provides an illustration of the main issue. In normal cases, after executing S2, coroutine-1 switches to coroutine-2, forcing the CPU to work continuously. This interactive execution allows the utilization of CPU and I/O resources in the pipeline. However, when they encounter the fragments of S2, both coroutine-1 and coroutine-2 swiftly complete their execution and then block waiting for S3 to complete. This results in idle and wasted CPU cycles.

C. Coroutine-based Method

Our optimization is twofold. First, it eliminates unnecessary CPU waiting time caused by the fragments. To address the problem, we separate the write operation from the original compaction process using a dedicated flush coroutine. The effect of our method is shown in Figure 4(d). In this case, as S2 will no longer be clipped by S3, the total compaction time is reduced. Second, we schedule coroutines to prevent bursty I/O contention. As the figure shows, S3 and S1 are almost scheduled to run alternately. This reduces the latency caused by I/O contention.

To achieve the above purpose, the coroutine-based method should support three parts. (i) a flush coroutine that is specialized to execute S3, (ii) a coroutine scheduling policy that controls I/O operations, (iii) a compaction splitter that divides the compaction task into the coroutine-level subtasks. Next, we will introduce how to implement them in detail.

Flush coroutine. We propose a flush coroutine that is specialized to execute S3. By removing all S3 from the initial compaction process, it avoids S3 cutting off S2. Given that the size of the read buffer is the only element influencing S1's trigger interval, S2 will not be further fragmented. Considering that a certain amount of delay in executing S3 does not logically prevent accessing the correct data because an LSM-tree typically allows old and new data to coexist, which makes the newly generated SSTables not accessible until all compaction tasks have been finished.

Inside a work thread, there is one flush coroutine and a number of compaction coroutines (executing S1 and S2). As all S3s are executed by the flush coroutine, we control their execution to tune the I/O pressure. By using the coroutine scheduling approach below, the I/O pressure in the compaction process can be even.

Coroutine scheduling. To properly handle I/O operations, we propose a coroutine scheduling policy. An increase in the quantity of concurrent I/O requests causes an increase in I/O latency. The solution is to adaptively execute I/O operations while I/O pressure is low and to restrict the amount of I/O operations when the pressure becomes high, thus making the I/O operations more even. When I/O device usage is low, our scheduling method wakes the flush coroutine to start I/O requests. When I/O pressure is high, however, we avoid waking the coroutines that are about to start execution of $S3$.

Our system does this by keeping track of the number of concurrent I/O operations. Let q_{comp} be the real-time record of the number of I/O operations for compactions, and let q_{cli} be the number of I/O actions for the current reads happening when accessing SSD. Because of this, the flush coroutine permits a maximum number of I/O requests of

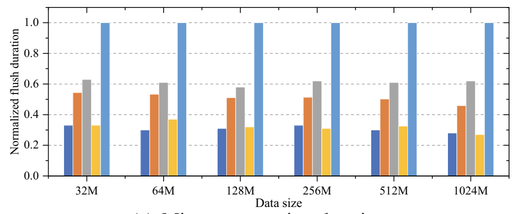
$$q_{flush} = \max\{q - q_{comp} - q_{cli}, 0\},$$

where q is a user-set value(e.g. 8) indicating the maximum number of concurrent I/O operations allowed, which depends on the performance of the I/O device. If $q_{flush} > 0$ it allows for the flush coroutine to immediately trigger its execution of S3. With this approach, I/O resources are used much more effectively, and the rise in I/O latency caused by many concurrent I/O operations is reduced.

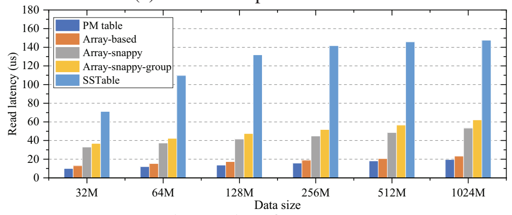
Compaction tasks. Coroutines in one thread are unable to utilize the multi-cores. Therefore, we further created a compaction task manager to divide the original compaction tasks into coroutine-compatible subtasks and reduce the compaction time by utilizing multi-cores. To take full advantage of multi-core, we assign c worker threads to c physical cores, and we also allow the user to select the number of physical cores to use. Inside a work thread, there is one flush coroutine and k compaction coroutines. Increasing k can improve CPU usage, but it will put more demand on I/O. To ensure that the I/O pressure is within a reasonable range, we set k as $k = \max\{\lfloor q/c \rfloor, 1\}$.

VI. Experiment

In this section, we evaluated *PM-Blade* to see the performance benefits of the proposed methods. We also demonstrated how *PM-Blade* improves the performance of LSM-tree in the Meituan's workload.



(a) Minor compaction duration



(b) Read performance

Fig. 6. Minor compaction duration and read performance of PM table

A. Experimental setup

Comparing Systems. (i) *PM-Blade*. We implemented all the mentioned characteristics based on the LSM-tree structured storage developed by Meituan's database group. *PM-Blade* allocated 80 GB PM for level-0, and all partitions in *PM-Blade* share the PM space. The size of memtable is set to 64MB, which is the same as RocksDB. (ii) RocksDB. RocksDB represents the conventional LSM-tree on a DRAM-SSD hierarchy, it triggers major compactions when the number of level-0 tables reaches 4, and uses threads for multiple compaction tasks. (iii) MatrixKV [9]. MatrixKV is a key-value store implemented based on RocksDB, it reorganizes an 8 GB level-0 in PM by default, and to be fair, we also added an 80 GB PM configuration in the following test cases. Its memtable is also set to 64MB.

Benchmarks. We used three benchmark tools for evaluations. (i) To evaluate the performance of PM level-0 and coroutine-based compaction, we developed a benchmark tool benchmark_kv based on db_bench, the micro-benchmark released with RocksDB. In addition to basic key-value operations, we added support for creating record tables and index tables on key-value stores. (ii) To evaluate the overall performance, we used the benchmark tool YCSB [31]. (iii) To see *PM-Blade*'s performance in a database scenario, we tested a scenario formed by a real-world workload of Meituan.

Hardware Environment. All experiments were run on a testing machine with two Intel(R) Xeon(R) Gold 6240R CPU@2.40GHz processors and 128 GB of memory. We used two storage devices in our experiments: a 1TB SSD to store SSTables; one chip of 128 GB Intel Optane DC Persistent Memory. The operating system is CentOS 7.8.

B. Evaluation of the PM-based level-0

Evaluation of the design of PM table. We evaluated the performance of minor compaction and read operations of different structures of PM table. We also tested alternative compression approaches on the array-based structure. Five alternatives were used in the comparison. (i) PM table uses our proposed structure and prefix compression. (ii) Array-

TABLE IV
SPACE RELEASED BY INTERNAL COMPACTION

Data skew	0.0	0.2	0.4	0.6	0.8	1.0
Space released (GB)	11.6	11.7	12.1	12.8	14.1	16.2

TABLE V
COMPACTION DURATION

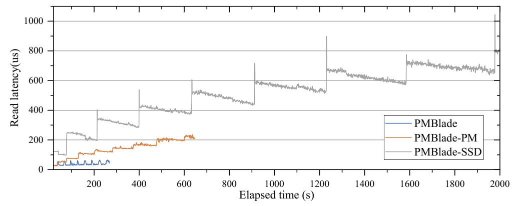
Value size	512B	1KB	4KB	16KB	64KB
PMBlade	2.1 s	1.9 s	1.9 s	1.6 s	1.4 s
PMBlade-SSD	4 s	3.7 s	3.4 s	2.9 s	2.8 s

based uses a simple array-based structure without compression [9]. (iii) Array-snappy uses an array-based table and the snappy [28] algorithm for compression (snappy is adopted by RocksDB and many other key-value databases); it compresses each key-value pair separately. (iv) Array-snappy-group is similar to Array-snappy, except that it compresses multiple (eight) key-value pairs as a group using snappy. (v) SSTable is the exact SSTable structure used in RocksDB. We inserted different amounts of data, and then manually triggered minor compactions to observe their duration, then, we read a million items at random. We used index tables of real-world workloads to build data set in this experiment, in which the index column size is 120 bytes.

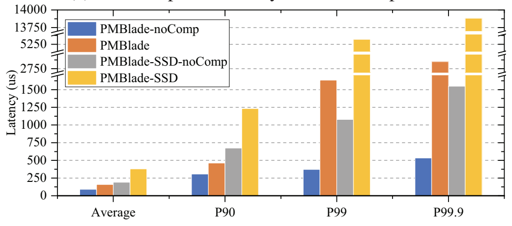
Figure 6(a) shows the normalized performance of different structures in minor compaction. PM table is the fastest approach. It outperforms Array-based and SSTable by around 40% and 70% in latency, respectively. Array-snappy fails to improve the performance of minor compaction duration, as the snappy compression it uses is expensive. Array-snappy-group is faster than Array-based by about 40%, because group compression can reduce the number of compression operations, thus reducing the compression overhead. In the meantime, group compression has a higher compression rate, which further reduces the overhead of PM writes. As the compression ratios for PM table stayed almost the same in all the tests, the time-saving rate were consistent.

Figure 6(b) shows the impact of compression on read performance. We found that, despite the overhead of decompression, PM table's read latency is lower than Array-based. When the data size is 32MB, the read latency of PM table is 22% less than that of Array-based, and the gap between them decreases as the data size increases. This can be attributed to the design of our compressed structure, which helps reduce the number of random accesses to the PM. We can also see that PM table can reduce the latency by up to 89% when compared against SSTable, due to the good performance of PM. The read latency of Array-snappy is about 2.3x higher than that of Array-based, because Array-snappy needs to decompress the key-value pair before accessing each key, which is time-consuming. Array-snappy-group performs worse than Array-snappy, because Array-snappy-group needs to decompress a group before accessing the key, which is more expensive.

Internal compaction performance. We first tested how much PM space can be freed up by internal compaction. Next, we tested the duration of internal compaction. Finally, we tested the impact of internal compaction on read performance. We used three configurations of *PM-Blade* in the experiments. (i) PMBlade, which performs internal compaction;



(a) Read improvement by internal compaction



(b) Latency with different compaction

Fig. 7. Read performance affected by internal compaction

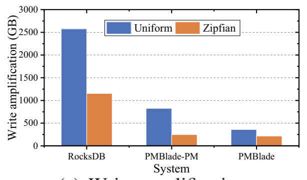
(ii) PMBlade-PM, which uses PM as level-0 but does not perform internal compaction; (iii) PMBlade-SSD, which uses traditional SSD-based level-0.

Table IV shows the space that can be released by internal compaction under various skewnesses of data. We used the update-only workload to write 20 GB data to *PM-Blade* before triggering internal compaction manually. We can see that the amount of space released by internal compaction increases as the data skewness increases. When the skewness is 1, internal compactions can free about 80% of used PM space. This is expected. The more skewed the workload, the more concentrated the updates, resulting in more redundancy in the unsorted PM tables.

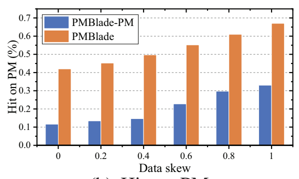
Table V shows the duration of the internal compaction. We inserted 1 GB of data and triggered compaction manually to measure the duration. We can see that the duration of internal compaction is much smaller than that of the SSD-based compaction because PM is more performant than SSD. When the value size is 64KB, the duration of the internal compaction is 50% of that of the SSD-based compaction.

Figure 7(a) shows the read latency of level-0 of different approaches. We performed 10 million operations (50% read and 50% write) and recorded the read latency. We can see that the latency of PMBlade always stays low, while the latency of PMBlade-PM and PMBlade-SSD increases significantly as the amount of data increases. Compared to PMBlade-PM, PMBlade reduces the read latency by a maximum of 82%. This is because internal compaction can mitigate the impact of level-0's read amplification. We also observed that internal compaction will affect ongoing read and write operations. It can increase the read latency by about 70%. Even in such a case, the latency of PMBlade is still far lower than that of PMBlade-SSD and PMBlade-PM.

Figure 7(b) provides another perspective on the impact of internal compaction on the ongoing reads. We first inserted 1 GB of data with 1 KB values into PMBlade and PMBlade-SSD. We then manually triggered internal compaction and traditional compaction respectively, and at the same time tested



(a) Write amplification



(b) Hit on PM

Fig. 8. Write amplification and proportion of access on PM with our strategy

the read latency during compaction. In addition, we also tested two cases in which compaction is not triggered, denoted by PMBlade-noComp and PMBlade-SSD-noComp. We can see that the average latency and the 99.9^{th} percentile latency of PMBlade are 1.7x and 5.3x higher than PMBlade-noComp respectively. Despite this, the average latency and the 99.9^{th} percentile latency of PMBlade are only 23% and 21% of those of PMBlade-SSD, mainly due to PM's high performance. Therefore, although internal compaction has an impact on ongoing reads, it is much smaller than the impact of traditional compaction on the ongoing reads on the SSD.

Performance improved by our compaction models. We tested the system's mitigation of write amplification after using our compaction model, and how much hot or warm data our compaction model keeps on PM. We used three systems for comparison: (i) RocksDB, using the default configuration of RocksDB. Since PMBlade-SSD and PMBlade-PM have the same structure and compaction approach, which leads to the same write amplification, we used RocksDB as a comparison. (ii) PMBlade-PM, using 80 GB PM for level-0 and the conventional level-0 compaction strategy. That is, when the number of PM tables reaches the threshold, the whole level-0 will be compacted to level-1. (iii) PMBlade, using 80 GB PM for level-0, and internal compaction and compaction models are used in this configuration.

First, we tested the mitigation of write amplification with our work. We first inserted 200 GB of data with different distributions, the value size is 1KB. Then we recorded the write amplification of each test case. Figure 8(a) shows the write amplification of the three systems under different distributions. Under the uniform distribution, the write amplification of PMBlade is 359GB, including 201 GB on PM, and 158 GB for the SSD, which is greatly lower than RocksDB and 43% of PMBlade-PM. The write amplification of RocksDB is 2573 GB, and PMBlade-PM is 825 GB. This is because our compaction model compacts most of the redundant data through internal compactions, which reduce the major compaction from level-0 to level-1. Although internal compactions are more frequent than major compactions, the write amplification produced by each internal compaction is far less than that of each major compaction. That is why PMBlade can outperform other systems in write amplification.

Then we tested our compaction model's ability to leave the hot or warm data on PM level-0. We used PMBlade-PM and PMBlade as a comparison. We used 50% read and 50% write workload to test. Figure 8(b) shows the proportion of read requests hitting PM under different data skews. With the increase of data skew, the hit rate of PM level-0 is increasing. When the data is not skewed, the hit rate of using our compaction model is 34% higher than that of not using our compaction model. This is because our compaction model will only compact the cold data into level-1, ensuring that the hot or warm data can remain on the PM. The traditional strategy will periodically compact the whole level-0 into level-1, which causes the PM to fail to retain the hot or warm data.

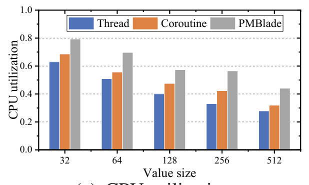
C. Evaluation of coroutine-based compaction

To show how our coroutine-based compaction can improve the utilization of system resources and shorten the compaction time, we used three configurations to compare: (i) Thread. Use multithreading to schedule tasks, which is also used as the default scheduling method of RocksDB. (ii) Coroutine. Use the basic coroutine task scheduling policy. This policy will switch to another coroutine when one coroutine encounters I/O waiting. (iii) PMBlade. Our coroutine-based compaction, including the redesigned coroutine scheduling strategy and flush coroutine. We compared their compaction performance under different value sizes. The smaller the value, the higher the CPU overhead, and the larger the value, the higher the I/O overhead. We first inserted the 2 GB data into the system and then triggered the major compaction. We set the concurrency of compaction tasks to 4, and use two CPU cores to execute these tasks. The maximum I/O concurrency is 4.

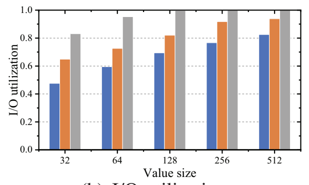
CPU utilization. Figure 9(a) shows the utilization of CPU resources under three configurations, and PMBlade is much better than other configurations. When the value size is 256, PMBlade is 23% higher than Thread and 14% higher than Coroutine, Coroutine is 9% higher than Thread. This is because the design of the flush coroutine reduces the CPU idle caused by fragmented time. It can be seen that a simple coroutine scheduling strategy can improve the utilization of the CPU, but is limited by fragmented time.

I/O device utilization. Figure 9(b) shows their I/O utilization. When the value size is 32, the PMBlade's I/O utilization is 35% higher than that of Thread and 18% higher than that of Coroutine. When the value size is greater than 128, the PMBlade's utilization of the I/O device is close to 100%. This is because our coroutine-based scheduling policy wakes up the flush coroutine to perform write operations when the I/O device is idle. While improving the I/O utilization, PMBlade does not increase the pressure of I/O devices. Figure 9(c) shows that the I/O latency of the PMBlade is the lowest during the compaction process. When the value size is 512, the latency of the PMBlade is 66% of the Thread. This is because our coroutine scheduling strategy only wakes up the flush coroutine to perform write operations when the I/O utilization is low, avoiding the problem of excessive I/O concurrency.

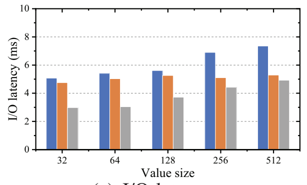
Major compaction duration. Figure 9(d) shows the compaction duration under the three configurations. As the value size increases, the compaction duration of all systems increases. The compaction duration of PMBlade is the shortest of the three systems. When the value size is 64, the compaction duration of PMBlade is 71% of that of Thread and 80% of that of Coroutine. This is attributed to the flush coroutine and our



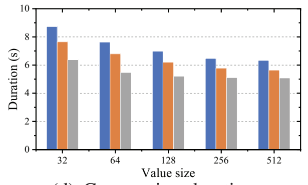
(a) CPU utilization



(b) I/O utilization

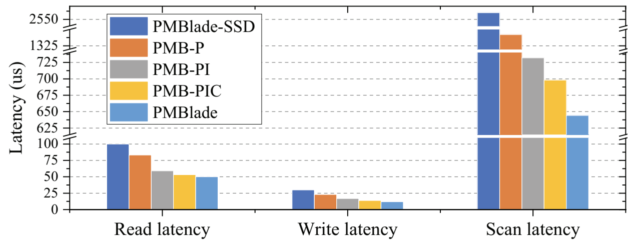


(c) I/O latency

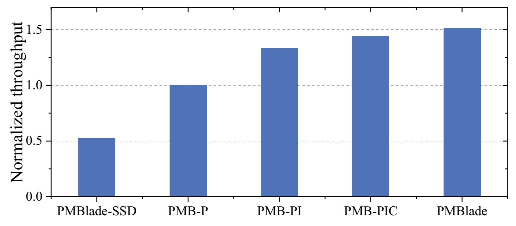


(d) Compaction duration

Fig. 9. Performance with our coroutine-based compaction



(a) Latency



(b) Throughput

Fig. 10. Ablation study on the real-world workload

coroutine scheduling strategy, which effectively reduce the idle time of CPU and I/O devices, thereby enhancing the utilization of system resources and decreasing the compaction duration.

D. Ablation study

We conducted an ablation study on a real-world workload of Meituan's online retail application to understand how much each of *PM-Blade*'s techniques contributes to the performance.

Workload Description. In the database of the online retail application, there are in total 10 tables, each containing about 10 columns. Read operations include both primary key queries and non-primary key queries. Most of the queries are index query. Indexes are created on frequently accessed columns, resulting in 3 indexes per table on average. To execute an index query, the system needs to obtain the row id through a scan operation, and then perform a point read to retrieve the target row. Therefore, recent data are more likely to be read. Write operations are mainly made up of inserts and updates. When a user creates a new order, the client will insert rows into multiple tables. This results in a number of sequential writes and random writes. Furthermore, each order will approximately write about 100 KB of data into the database. As the order progresses, the status of the transaction needs to be updated in time, which results in many updates.

Ablation study. To make the ablation study, we used five configurations of *PM-Blade*. (i) PMBlade-SSD, which does not deploy any technique in this paper and does not use PM either. (ii) PMB-P, which puts level-0 on PM and uses array-based tables as the storage structure. (iii) PMB-PI, which uses array-based tables on PM and performs internal compaction using the cost-based compaction strategy. (iv) PMB-PIC, which uses compressed PM table and performs internal compaction using the cost-based compaction strategy. (v) PM-Blade, which additionally deploys coroutine-based compaction on top of PMB-PCI (i.e., all techniques are applied). For the experiments, we first inserted 200 GB of data into the database and then executed one million transactions to observe their execution latency.

Figure 10(a) shows the contribution of each technology to the end-to-end latency of read, scan, and write. Regarding read latency, PMBlade achieved a 40% reduction compared to PMB-P. Internal compaction, compression and coroutine-based compaction contributed a 29% reduction, a 7% reduction and a 4% reduction, respectively. The internal compaction reduces the read amplification in level-0, resulting in the most read performance improvement. By preserving more hot and warm data in PM, our compaction model can significantly enhance the read performance too.

Regarding write latency, PMBlade achieved a 48% reduction compared to PMB-P. Internal compaction, compression and coroutine-based compaction contributed a 27%, a 13% and an 8% reduction respectively. This is because a large level-0 and the internal compaction can significantly reduce the write amplification on the SSD in major compaction. Our compression-based PM table can also speed up minor compactions, which helps release DRAM space faster.

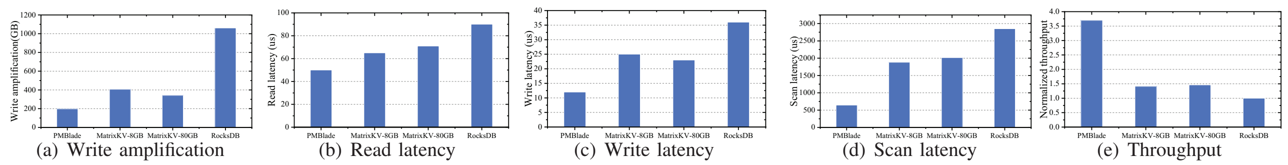
As to the latency of scan operations, PMBlade achieved a 54% reduction compared to PMB-P. Internal compaction alone contributed to a 43% reduction. Compared to PMBlade-SSD, PMB-P also reduced the latency by 49%. This is because PM's high performance accelerates the scan operations. Internal compaction reduces read amplification in level-0, thus reducing invalid reads on PM tables significantly.

Figure 10(b) shows the throughputs. PMBlade outperforms PMB-P by 51%. Internal compaction contributed a 33% improvement, while compression and coroutine-based compaction contributed 11% and 7% respectively. In summary, we can conclude that a large level-0 on PM can significantly improve the read and write performance of an LSM-tree, and PM table and coroutine-based compaction also play important roles in improving the overall performance.

E. Comparing with other systems

To understand the overall performance of *PM-Blade*, we compared the performance of PMBlade, RocksDB and MatrixKV under YCSB and Meituan's real-world workload.

YCSB performance. We first evaluated three systems with YCSB [31], which includes 7 workloads named from A to



(a) Write amplification (b) Read latency (c) Write latency (d) Scan latency (e) Throughput

Fig. 11. Performance on the real-world workload

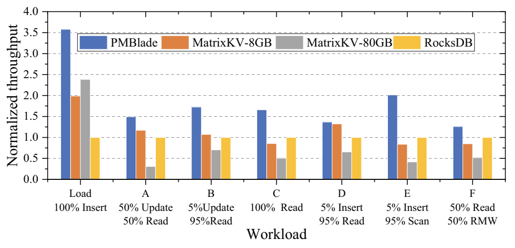


Fig. 12. Normalized throughput under different YCSB workloads

F often used to comprehensively evaluate the performance of KV stores. The default configuration of MatrixKV is 8GB. Considering the PM's space is far less than that of PMBlade, we added a configuration of using 80 GB PM for MatrixKV (MatrixKV-80GB). For the experiments, we first wrote a 200 GB dataset with 1KB value for loading, then evaluated workloads A-F with one million KV items respectively.

Figure 12 shows their throughput. For the Load workload, the throughput of PMBlade is 3.5x and 1.8x faster than that of RocksDB and MatrixKV-8GB, it is profit from our large PM level-0 as the write buffer, which reduces the I/O overhead for the SSD. Although MatrixKV-80GB also uses large PM, its complex level-0(matrix container) requires additional construction overhead, which slows down minor compaction and hinders the high performance of PM.

Workloads B to E are dominated by read. For workload E, PMBlade is 2.0x and 2.4x faster than RocksDB and MatrixKV. This is because internal compaction alleviates level-0's read amplification problem, and our flat structure is friendly to scan. MatrixKV proposed a cross-hint search to speed up read, but its compaction strategy does not keep hot or warm data on PM, and it does not fully utilize PM's benefits.

Workloads A and F are dominated by read and write. For workload A, PMBlade is 1.5x and 1.3x faster than RocksDB and MatrixKV-8GB. This is because our compaction strategy compacts most of the data in PM, greatly reducing the number of major compactions from level-0 to level-1, easing write amplification and improving read performance at the same time. Additionally, PMBlade leave the hot or warm data on PM, significantly decreasing the amount of data retrieved from the SSD, which is crucial in lowering read latency.

Performance on real-world workload. Next, we further compared the three systems with Meituan's real-world workload and then shown the evaluation results in detail. The workload is the same as we used in subsection VI-D

Figure 11 shows the performance improvement of our work in real-world workload in detail. Figure 11(a) shows the write amplification of the four systems. The write amplification of the PMBlade is 197GB, including 125GB on PM and 72GB on SSD, which is only 18% of that of RocksDB. This is because our compaction model absorbs most of the write amplification through internal compaction, which greatly reduces the write amplification caused by major compaction. The write amplification of MatrixKV-8 is 2.1x larger than that of PMBlade, as it only relies on the reduction of the LSM-tree layers to alleviate write amplification.

Figure 11(b) shows the read latency of the four systems. PMBlade has the lowest read latency, as the internal compaction and the compaction model managed to minimize level-0's read amplification. MatrixKV improves the read performance by building a complex index structure on level-0. However, from the results, its effect is not as good as that of internal compaction. Figure 11(c) shows the write latency. The write latency of PMBlade is 33% of RocksDB and 48% of MatrixKV-8GB. This is because our large PM level-0 and compaction model can reduce the write amplification on SSD, thus releasing I/O for writing logs, and our PM table can also improve the flush performance. Figure 11(d) shows the scan latency. PMBlade also performed the best. The scan latency of PMBlade is 34% of that of MatrixKV-8GB and 22% of that of RocksDB. This is because our flat structure can reduce the cost of range queries, and our compaction model can reduce the impact of level-0's read amplification. Figure 11(e) shows the normalized throughput of the four systems. The throughput of PMBlade is far better than that of other systems, which is 2.6x and 2.5x faster than MatrixKV-8GB and MatrixKV-80GB respectively, and 3.7x faster than RocksDB.

VII. Conclusion and Future Work

In this paper, we present *PM-Blade*, an LSM-tree structured storage augmented with persistent memory. To achieve a holistic optimization on both read and write performance, we used PM as level-0 of the LSM-tree and redesigned the internal structure and the function of level-0. Through *PM-Blade*, we demonstrated that persistent memory can be used to significantly improve the performance of LSM-tree based database storage. In our future work, we will explore applying *PM-Blade* 's approach to other high-capacity memory devices, such as CXL expanded memory.

Acknowledgments

This work was sponsored by the National Science Foundation of China under grant number 61972149. This research was also supported by Meituan. We are grateful to Huanqing Dong and Kai Li for their valuable contributions to this work.

References

[1] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, W. Wei, C. Liu, J. Zhang, J. Li, X. Wu, L. Song, R. Sun, S. Yu, L. Zhao, N. Cameron, L. Pei, and X. Tang, "Tidb: A raft-based HTAP database," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3072–3084, 2020.

[2] G. Huang, X. Cheng, J. Wang, Y. Wang, D. He, T. Zhang, F. Li, S. Wang, W. Cao, and Q. Li, "X-engine: An optimized storage engine for large-scale e-commerce transaction processing," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 651–665. [Online]. Available: https://doi.org/10.1145/3299869.3314041

[3] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi, H. Yu, B. Liu, Y. Pan, B. Yin, J. Chen, and Q. Xu, "Oceanbase: A 707 million tpmc distributed relational database system," *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3385–3397, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3385-xu.pdf

[4] C. Luo and M. J. Carey, "Lsm-based storage techniques: a survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, 2020.

[5] S. Kannan, N. Bhat, A. Gavrilovska, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redesigning lsms for nonvolatile memory with novelsm," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. Reed, Eds. USENIX Association, 2018, pp. 993–1005.

[6] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. Choi, "SLM-DB: single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*, A. Merchant and H. Weatherspoon, Eds. USENIX Association, 2019, pp. 191–205.

[7] J. Lindström, D. Das, T. Mathiasen, D. Arteaga, and N. Talagala, "NVM aware mariadb database system," in *IEEE Non-Volatile Memory System and Applications Symposium, NVMSA 2015, Hong Kong, China, August 19-21, 2015*. IEEE, 2015, pp. 1–6.

[8] B. Yan, X. Cheng, B. Jiang, S. Chen, C. Shang, J. Wang, K. Huang, X. Yang, W. Cao, and F. Li, "Revisiting the design of lsm-tree based OLTP storage engine with persistent memory," *Proc. VLDB Endow.*, vol. 14, no. 10, pp. 1872–1885, 2021.

[9] T. Yao, Y. Zhang, J. Wan, Q. Cui, L. Tang, H. Jiang, C. Xie, and X. He, "Matrixkv: Reducing write stalls and write amplification in lsm-tree based KV stores with matrix container in NVM," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, A. Gavrilovska and E. Zadok, Eds. USENIX Association, 2020, pp. 17–31.

[10] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," *login Usenix Mag.*, vol. 45, no. 3, 2020.

[11] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "Pebblesdb: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 497–514.

[12] F. Pan, Y. Yue, and J. Xiong, "dcompaction: Speeding up compaction of the lsm-tree via delayed compaction," *J. Comput. Sci. Technol.*, vol. 32, no. 1, pp. 41–54, 2017.

[13] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, "Building efficient key-value stores via a lightweight compaction tree," *ACM Trans. Storage*, vol. 13, no. 4, pp. 29:1–29:28, 2017.

[14] K. Huang, Z. Jia, Z. Shen, Z. Shao, and F. Chen, "Less is more: De-amplifying i/os for key-value stores with a log-assisted lsm-tree," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2021, pp. 612–623.

[15] H. Amur, D. G. Andersen, M. Kaminsky, and K. Schwan, "Design of a write-optimized data store," Georgia Institute of Technology, Tech. Rep., 2013.

[16] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, A. D. Brown and F. I. Popovici, Eds. USENIX Association, 2016, pp. 133–148.

[17] H. H. W. Chan, Y. Li, P. P. C. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in KV storage via hashing," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, H. S. Gunawi and B. Reed, Eds. USENIX Association, 2018, pp. 1007–1019.

[18] Y. Li, Z. Liu, P. P. C. Lee, J. Wu, Y. Xu, Y. Wu, L. Tang, Q. Liu, and Q. Cui, "Differentiated key-value storage management for balanced I/O performance," in *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, I. Calciu and G. Kuenning, Eds. USENIX Association, 2021, pp. 673–687.

[19] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka, "TRIAD: creating synergies between memory, disk and log in log structured key-value stores," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds. USENIX Association, 2017, pp. 363–375.

[20] N. Dayan, M. Athanassoulis, and S. Idreos, "Optimal bloom filters and adaptive merging for lsm-trees," *ACM Trans. Database Syst.*, vol. 43, no. 4, pp. 16:1–16:48, 2018.

[21] Niv Dayan and Manos Athanassoulis and Stratos Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 79–94.

[22] X. Wu, Y. Xu, Z. Shao, and S. 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, July 8-10, Santa Clara, CA, USA*, S. Lu and E. Riedel, Eds. USENIX Association, 2015, pp. 71–82.

[23] W. Zhang, X. Zhao, S. Jiang, and H. Jiang, "Chameleondb: a key-value store for optane persistent memory," in *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds. ACM, 2021, pp. 194–209.

[24] Oana Balmau and Florin Dinu and Willy Zwaenepoel and Karan Gupta and Ravishankar Chandhiramoorthi and Diego Didona, "SILK: preventing latency spikes in log-structured merge key-value stores," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds. USENIX Association, 2019, pp. 753–766.

[25] O. Balmau, F. Dinu, W. Zwaenepoel, K. Gupta, R. Chandhiramoorthi, and D. Didona, "SILK+ preventing latency spikes in log-structured merge key-value stores running heterogeneous workloads," *ACM Trans. Comput. Syst.*, vol. 36, no. 4, pp. 12:1–12:27, 2020.

[26] Z. Zhang, Y. Yue, B. He, J. Xiong, M. Chen, L. Zhang, and N. Sun, "Pipelined compaction for the lsm-tree," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 2014, pp. 777–786. [Online]. Available: https://doi.org/10.1109/IPDPS.2014.85

[27] W. Kim, C. Park, D. Kim, H. Park, Y. Choi, A. Sussman, and B. Nam, "Listdb: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 161–177.

[28] (2022) Snappy. [Online]. Available: https://github.com/google/snappy

[29] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[30] O. Goldreich, S. Micali, and A. Wigderson, "How to prove all np statements in zero-knowledge and a methodology of cryptographic protocol design," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1986, pp. 171–185.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, J. M. Hellerstein, S. Chaudhuri, and M. Rosenblum, Eds. ACM, 2010, pp. 143–154.