MatrixKV: Reducing Write Stalls and Write Amplification in LSM-Tree Based KV Stores with Matrix Container in NVM

摘要

基于 LSM 树（Log-Structured Merge-Tree）的流行键值存储系统因写放大（write amplification）和写停滞（write stalls）问题而遭受次优和不可预测的性能表现，这些问题导致应用性能周期性地降至几乎为零。我们的初步实验研究揭示，（1）写停滞主要源于在 LSM 树的前两层（L0-L1）之间的合并操作中涉及的大量数据量；（2）随着 LSM 树深度的增加，写放大现象也随之增加。现有的研究主要集中在减少写放大上，而只有少数研究致力于缓解写停滞问题。我们利用非易失性内存（NVM）来解决这两个限制，并复现了 MatrixKV——一种基于 LSM 树的新型键值存储（KV store），适用于具有多层 DRAM-NVM-SSD 存储的系统。我们基于 RocksDB 实现了 MatrixKV，并在搭载英特尔最新 3D Xpoint NVM 设备 Optane DC PMM 的混合 DRAM/NVM/SSD 系统上对其进行了评估。评估结果显示，与 RocksDB 和最新的基于 LSM 的 KVS NoveLSM 相比，MatrixKV 在相同的 NVM 量下实现了 5 倍和 1.9 倍更低的 99 分位长尾延迟，以及 3.6 倍和 2.6 倍更高的随机写入吞吐量。

关键词： LSM 树；持久内存；键值存储

1 引言

持久化键值存储在支持现代数据中心中大量应用方面日益重要。在写密集型场景中，日志结构合并树（LSM 树）是持久化键值（KV）存储的核心索引结构，如 RocksDB 、LevelDB、HBase 和 Cassandra 。考虑到在流行的 OLTP（在线事务处理）工作负载中随机写入很常见，随机写入的性能，特别是持续的或突发的随机写入，对用户来说是一个严重的问题 [1]。本文将 KV 存储的随机写入性能作为主要关注点。流行的 KV 存储部署在具有 DRAM-SSD 存储的系统上，旨在利用快速的 DRAM 和持久的 SSD 提供高性能的数据库访问。然而，诸如单元大小、功耗、成本和 DIMM 插槽可用性等限制阻止了通过增加 DRAM 大小来进一步提高系统性能 [3]。因此，在混合系统中利用非易失性内存（NVMs）被广泛认为是一种提供更高系统吞吐量和更低延迟的有希望的机制。

LSM 树（日志结构合并树）通过多个指数级增长的层级存储键值（KV）项，例如从 L_0 到 L_6。为了更好地理解基于 LSM 树的键值存储，我们对流行的 RocksDB 进行了实验评估，该系统采用传统的 DRAM-SSD 存储，并观察到两个挑战性问题及其根本原因。首先，写停

滞导致应用吞吐量周期性地降至几乎为零，从而造成性能的剧烈波动和长尾延迟。系统吞吐量的低谷表明了写停滞的存在。写停滞引起高度不可预测的性能，并降低了用户体验的质量，这与 NoSQL 系统预期的可预测和稳定性能的设计目标相悖 [21]。此外，写停滞显著延长了请求处理的延迟，造成了高尾延迟 [6]。我们的实验研究表明，写停滞的主要原因是每次 $L_0 - L_1$ 合并处理的大量数据。由于 $L_0 - L_1$ 层未排序（L_0 中的文件与键范围重叠），L_1 层的合并操作涉及这两个层级的几乎所有数据。这种全对全的合并占用了 CPU 周期和 SSD 带宽，从而减慢了前台请求的处理速度，导致了写停滞和长尾延迟。其次，写放大（WA）降低了系统性能和存储设备的耐久性。WA 与 LSM 树的深度直接相关，因为由于数据集更大而导致的树结构更深，增加了合并操作的次数。尽管大量研究致力于减少 LSM 树的写放大 [19]，但只有少数已发表的研究关注缓解写停滞 [6]。我们的研究旨在同时应对这两个挑战。

针对这两个挑战及其根本原因，我们复现了 MatrixKV，这是一种针对具有 DRAM-NVM-SSD 存储系统的基于 LSM 树的键值存储。MatrixKV 的设计原则是利用 NVM 来为 L_0 和 L_1 构建更经济、更精细粒度的合并操作，以及减少 LSM 树的深度以缓解写放大。MatrixKV 的关键技术总结如下：

- **矩阵容器**：MatrixKV 中的矩阵容器通过接收器和压缩器在 NVM 中管理 LSM 树未排序的 L_0 层。接收器采用并保留从 DRAM 刷新的 MemTable，每行一个 MemTable。压缩器选择并合并 L_0 层中具有相同键范围的数据子集到 L_1 层，每次合并处理一个列。
- **列合并**：列合并是 L_0 与 L_1 之间的细粒度合并，它一次合并一个小键范围。列合并减少了写停滞，因为它处理有限的数据量，并迅速释放 NVM 中的列，以便接收器接受从 DRAM 刷新的数据。
- **减少 LSM 树深度**：MatrixKV 增加了每个 LSM 树层级的大小，以减少层级数。因此，MatrixKV 减少了写放大，并提供了更高的吞吐量。
- **跨行提示搜索**：MatrixKV 为每个键分配一个指针，逻辑上对矩阵容器中的所有键进行排序，从而加速搜索过程。

2 一研究背景

在这一部分中，我们将介绍关于非易失性内存（NVM）、LSM 树、基于 LSM 的键值（KV）存储的相关工作的基础背景。

2.1 非易失性内存

服务提供商一直在追求更快的数据库访问速度。他们的目标是在不显著增加总体拥有成本的情况下，为用户提供更好的服务质量和体验。随着相变存储器 [8,17]、忆阻器 [23] 和 STT-MRAM [14] 等新型存储介质的出现和发展，采用非易失性内存（NVM）来增强存储系统成为了一种成本效益高的选择。NVM 具有字节寻址能力、数据持久性和高速度。它有望提供类似 DRAM 的性能、类似硬盘的持久性，并且在成本上远低于 DRAM 的更高容量 [9]。与固态硬盘（SSD）相比，NVM 预计将提供 100 倍更低的读写延迟和高达十倍的更高带宽 [2]。

非易失性存储器既可以作为通过 PCIe 接口访问的持久化块存储设备，也可以作为通过内存总线访问的主存储器。现有研究表明，前者只能实现边际性能改进，浪费了 NVM 的高性能。对于后者，非易失性存储器可以作为单级存储系统中取代或补充 DRAM [13]，也可以作为 NVM-SSD 系统或 DRAM-NVM-SSD 混合系统的一部分。特别是，采用 DRAM-NVM-SSD 存储的系统被认为是利用非易失性存储器的一种有前景的方式，原因有三：首先，预计 NVM 将在未来几年与大容量 SSD 并存。其次，与 DRAM 相比，NVM 的带宽仍然低 5 倍，读取延迟高 3 倍。第三，混合系统平衡了总体拥有成本和系统性能。因此，MatrixKV 专注于在 DRAM、NVM 和 SSD 的混合系统中高效使用 NVM 作为持久内存。

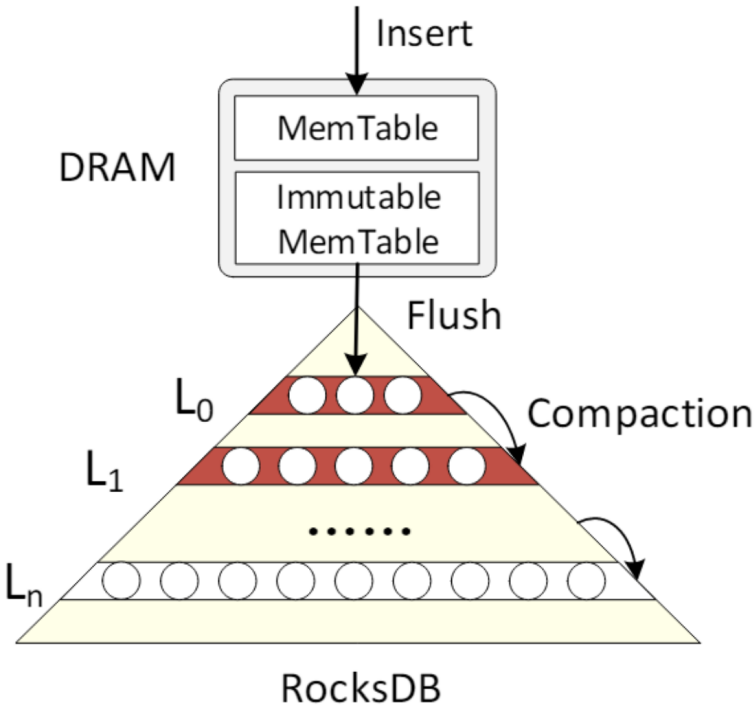

图 1. LSM 树示意图

2.2 LSM 树

LSM 树延迟并批量处理内存中的写请求，以利用存储设备的高顺序写入带宽。这里我们解释一种流行的 LSM 树实现，即广泛部署的基于 SSD 的 RocksDB。如图1所示，RocksDB 由 DRAM 组件和 SSD 组件组成。它还在 SSD 中有一个写前日志，用于保护 DRAM 中的数据免受系统故障的影响。为了处理写请求，首先在 DRAM 中通过两个跳表（MemTable 和不可变 MemTable）批量处理写入。然后，将不可变的 MemTable 刷新到 SSD 上的 L_0 层，生成排序字符串表（SSTables）。

为了实现快速刷新，L_0 层是未排序的，其中不同 SSTable 之间的键范围有重叠。SSTables 在 LSM 树的生命周期中从 L_0 层压缩到更深的层级（L_1，L_2...L_n）。压缩使得每个层级（除了 L_0）都是有序的，从而限制了读取和扫描的开销。进行压缩时，(1) 在 L_i 层选择一个 SSTable，并选择多个在 L_{i+1} 层有重叠键范围的 SSTable（称为重叠 SSTable）作为压缩数据。(2) 在 L_i 层选择落在这个压缩键范围内的其他 SSTable。(3) 将在步骤 (1) 和 (2) 中识别出的 SSTable 提取到内存中，以便合并和排序。(4) 重新生成的 SSTable 被写回到 L_{i+1} 层。由于 L_0 层是未排序的，且 L_0 层中的每个 SSTable 都涵盖广泛的键范围，因此 $L_0 - L_1$ 层的压缩需要反复

执行步骤（1）和（2），涉及两个层级的几乎所有 SSTable，导致大规模的全对全压缩。

为了处理读取请求，RocksDB 首先搜索 MemTable，接着是不可变 MemTable，然后按顺序搜索 L_0 至 Ln 层的 SSTable。由于 L_0 层的 SSTable 包含重叠键，因此在 L_0 层查找可能需要搜索多个文件。

2.3 基于键值存储的 LSM 树

对 LSM 树的现有改进包括：减少写放大 [11]、改进内存管理 [7]、支持自动调优 [10]，以及利用 LSM 树针对混合存储层次结构 [4]。其中，随机写性能是一个共同关注的问题，因为它受到压缩的严重阻碍。接下来，我们将讨论与我们工作最相关的研究，分为三类：减少写放大的、解决写停顿的，以及利用 NVM 的研究。

减少写放大（WA）：PebblesDB [20] 通过使用保护键来维护部分排序的层级，从而减轻了写放大。Lwc-tree 通过向 SSTable 附加数据并仅合并元数据来提供轻量级的压缩。WiscKey [19] 将键和值分离，压缩过程中只合并键，因而减少了写放大。键值分离的解决方案带来了垃圾回收和范围扫描的复杂性，且只对大值有利。LSM-trie [24] 通过基于哈希-范围的压缩来分摊压缩开销。VTtree [22] 使用额外的间接层来避免重新处理排序数据，但代价是造成了数据碎片化。TRIAD [5] 通过在内存、磁盘和日志之间创造协同作用来减少写放大。然而，几乎所有这些努力都忽视了性能的变化和写入停顿。

减少写停顿： SILK [6] 引入了一个 I/O 调度器，通过将刷新和压缩推迟到低负载时期、优先处理刷新和低层次压缩以及抢占式压缩，来减轻写停顿对客户端写入的影响。这些设计选择使得 SILK 在持续的写入密集型和长时间高峰负载下呈现出普通的写停顿。Blsm [21] 提出了一种新的合并调度器，称为“弹簧和齿轮”，用于协调多个层级的压缩。然而，它只限制了最大的写处理延迟，而忽略了大的排队延迟。KVell [18] 使 KV 项目在磁盘上无序，从而减少 CPU 计算成本，进而减轻基于 NVMe SSD 的 KV 存储的写停顿，但这对于使用通用 SSD 的系统不适用。

利用 NVM 改进 LSM 树：SLM-DB [15] 为配备 NVM-SSD 存储的系统提出了单层级 LSM 树。它在 NVM 中使用 B+ 树，为 SSD 上的单层级 LSM 树提供快速读取。这种解决方案伴随着维护 B+ 树和 LSM 树之间一致性的开销。MyNVM [12] 利用 NVM 作为块设备来减少基于 SSD 的 KV 存储中的 DRAM 使用。NoveLSM [16] 是用于具有 DRAM、NVM 和 SSD 混合存储系统的最先进的基于 LSM 的 KV 存储。NVMRocks 旨在创建一个 NVM 感知的 RocksDB，与 NoveLSM 类似，它采用了 NVM 上的持久可变 MemTable。然而，可变的 NVM MemTable 只在一定程度上减少了访问延迟，同时产生了更严重写停顿的负面效应。

由于我们为具有多层 DRAM-NVM-SSD 存储的系统构建 MatrixKV，并重新设计 LSM 树以利用 NVM 的高性能，因此 NoveLSM [16] 被认为与我们的工作最相关。我们在评估中以 NoveLSM 作为主要比较对象。此外，我们还评估了在基于 NVM 的系统上的 PebblesDB 和 SILK，因为它们是减少写放大或写停顿的最先进解决方案，但它们的原始设计并不是针对混合系统的。

3 本文方法

3.1 矩阵容器

MatrixKV 将 LSM 树的 L_0 层从 SSD 提升到 NVM，并将 L_0 重新组织成一个矩阵容器，以利用 NVM 的字节寻址能力和快速随机访问特性。矩阵容器是 LSM 树 L_0 层的数据管理结构。图2展示了矩阵容器的组织结构，它包括一个接收器和一个压缩器。

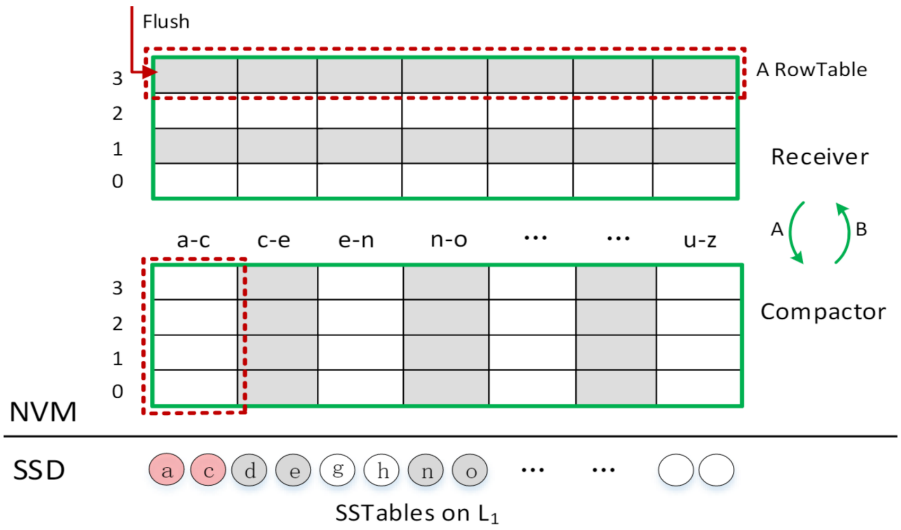

图 2. 矩阵容器的结构

接收器：在矩阵容器中，接收器接受并保留从 DRAM 刷新的 MemTable。每个这样的 MemTable 被序列化为接收器的单独一行，并组织为一个 RowTable。RowTables 以递增的序列号（即从 0 到 n）逐行附加到矩阵容器中。接收器的大小从一个 RowTable 开始。当接收器的大小达到其限制（例如，矩阵容器的 60%）且压缩器为空时，接收器停止接收刷新的 MemTable，并动态地转变为压缩器。同时，创建一个新的接收器来接收刷新的 MemTable。接收器向压缩器的逻辑角色变更不涉及数据迁移。

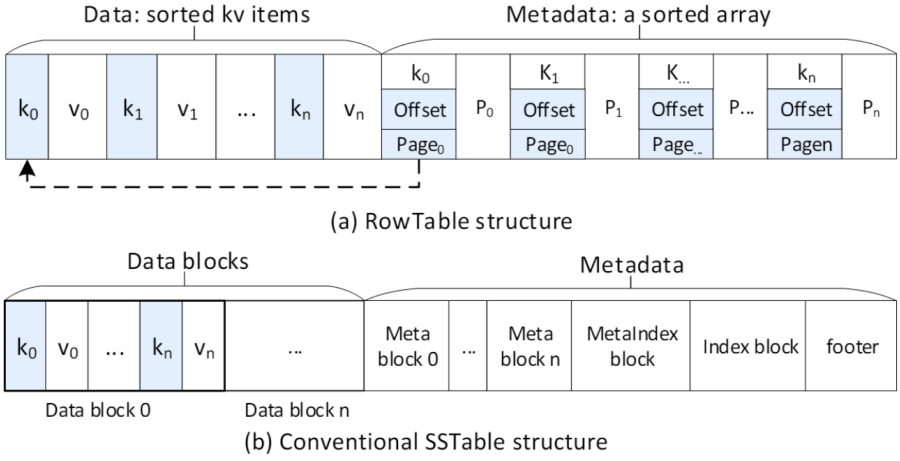

图 3. RowTable 的结构

RowTable：图3(a) 展示了 RowTable 的结构，包括数据和元数据。为了构建一个 RowTable，我们首先按键的顺序（与 SSTable 相同）序列化不可变 MemTable 中的 KV 项，并将它们存

储到数据区域。然后，我们为所有 KV 项建立一个排序数组作为元数据。每个数组元素维护键、页号、页内偏移量和一个向前指针（即 pn）。为了在 RowTable 中定位一个 KV 项，我们对排序数组进行二分搜索以获取目标键，并使用页号和偏移量找到其值。每个数组元素中的向前指针用于跨行提示搜索，这有助于提高矩阵容器内的读取效率。图3(b) 展示了 LSM 树中传统 SSTable 的结构。SSTables 是以块为基本单位组织的，与诸如 SSD 和 HDD 这类设备的存储单元相一致。相比之下，RowTable 以 NVM 页面作为其基本单位。除此之外，RowTables 与 SSTables 在元数据的组织上有所不同。因此，构建 SSTables 和 RowTables 的开销是相似的。

压缩器：压缩器用于从 L_0 层选择和合并数据到 SSD 中的 L_1 层，具有细粒度的合并能力。利用 NVM 的字节寻址能力和我们提出的 RowTables，MatrixKV 允许更经济的压缩，它可以合并 L_0 层特定键范围的数据与 L_1 层的一部分 SSTables，而不需要合并整个 L_0 层和整个 L_1 层。这种新的 $L_0 - L_1$ 压缩被称为列压缩。在压缩器中，KV 项通过逻辑列进行管理。列是键空间的一个子集，带有限量的数据，它是列压缩中压缩器的基本单位。具体来说，不同 RowTables 中落在某列压缩键范围内的 KV 项在逻辑上构成一个列。这些 KV 项的数量是列的大小，它并不是严格固定的，而是由列压缩的大小决定的阈值。

3.2 列合并

列压缩是一种细粒度的 $L_0 - L_1$ 压缩，每次只压缩一个列，即特定键范围内数据的一个小子集。因此，列压缩可以显著减少写停顿。列压缩的主要工作流程可以描述为以下七个步骤。(1) MatrixKV 将 L_1 的键空间分割成多个连续的键范围。由于 L_1 中的 SSTables 是排序的，每个 SSTable 由其最小键和最大键界定，因此 L_1 中所有 SSTable 的最小键和最大键形成一个排序的键列表。每两个相邻的键代表一个键范围，即一个 SSTable 的键范围或两个相邻 SSTable 之间的间隙。结果是，我们在 L_1 中有多个连续的键范围。(2) 列压缩从 L_1 中的第一个键范围开始。它选择 L_1 中的一个键范围作为压缩键范围。(3) 在压缩器中，上一步骤中被选中的 KV 项在多个行中同时被选中，落在压缩键范围内。具体来说，假设压缩器中有 N 个 RowTable，k 个线程并行工作以提取压缩键范围内的键。每个线程负责 N/k 个 RowTable。我们通过设置 $k = 8$ 来维持 NVM 上足够程度的并发访问。(4) 如果这个键范围内的数据量低于压缩的下限，则 L_1 中的下一个键范围加入。k 个线程在 N 个排序数组（即 RowTables 的元数据）中继续向前提取新键范围内的 KV 项。这个键范围扩展过程持续进行，直到压缩数据量达到一个介于下限和上限之间的大小（例如 $\frac{1}{2}AF \times S_{sst}$ 和 $AF \times S_{sst}$）。这两个界限保证了列压缩的适当开销。(5) 然后在压缩器中逻辑上形成一个列，即 N 个 RowTable 中落在压缩键范围内的 KV 项组成一个逻辑列。(6) 列中的数据与 L_1 中重叠的 SSTable 在内存中合并和排序。(7) 最后，重新生成的 SSTable 被写回到 SSD 上的 L_1。列压缩在 L_1 的下一个键范围和压缩器中的下一个列之间继续进行。列压缩的键范围在整个键空间中轮换，以保持 LSM 树的平衡。

3.3 减少 LSM 树深度

在传统的 LSM 树中，每个层级的大小限制通过一个放大因子 $AF = 10$ 来增长。LSM 树中的层级数量随着数据库中数据量的增加而增加。由于将 SSTable 压缩到更高层级会导致 AF 倍的写放大，因此整体写放大随 LSM 树中层级数量的增加而增加，即 $WA = n * AF$。因

此，MatrixKV 的另一个设计原则是减少 LSM 树的深度以减轻写放大。MatrixKV 通过在固定比例下增加每个层级的大小限制来减少 LSM 树的层级数量，从而使相邻层级的 AF 保持不变。结果是，对于从 L_1 和更高层级的压缩，将 SSTable 压缩到下一个层级的 WA 仍然是相同的 AF，但由于层级数量减少，整体 WA 得到了减少。

3.4 跨行提示搜索

在 MatrixKV 的 L_0 层中，每个 RowTable 都是排序的，且不同的 RowTable 在键范围上有重叠。为每个表构建布隆过滤器是减少搜索开销的一种可能解决方案。然而，这会带来构建过程的成本，并且对范围扫描没有益处。为了为 MatrixKV 提供足够的读取和扫描性能，我们构建了跨行提示搜索。

构建跨行指针：当我们为矩阵容器的接收器构建 RowTable 时，我们为元数据的排序数组中的每个元素添加一个向前指针（如图3所示）。具体来说，对于 RowTable i 中的一个键 x，向前指针索引前一个 RowTable $i-1$ 中的键 y，其中键 y 是不小于 x 的第一个键（即 $y \geq x$）。这些向前指针提供了提示，以逻辑上对不同行中的所有键进行排序，类似于分数级联。由于每个向前指针仅记录前一个 RowTable 的数组索引，因此向前指针的大小只有 4 字节。因此，存储开销非常小。

矩阵容器中的搜索过程：搜索过程从最新到达的 RowTable i 开始。如果 RowTable i 的键范围与目标键不重叠，我们跳转到其前一个 RowTable $i-1$。否则，我们对 RowTable i 进行二分搜索，以找到目标键所在的键范围（即由两个相邻键界定）。通过向前指针，我们可以在先前的 RowTable $i-1, i-2, \ldots$ 中缩小搜索区域，直到找到键为止。因此，无需遍历所有表格即可获取一个键或扫描一个键范围。跨行提示搜索通过显著减少搜索过程中涉及的表格和元素数量，提高了 L_0 的读取效率。

4 复现细节

4.1 代码结构

```
yunhao@a-X12DAi-N6:~/MatrixKV-master$ ls -l|grep ^d | awk '{print $1" "$2" "$3" "$4" "$5" "$9}'
drwxrwxr-x 2 yunhao yunhao 4096 buckifier
drwxrwxr-x 2 yunhao yunhao 4096 build_tools
drwxrwxr-x 2 yunhao yunhao 4096 cache
drwxrwxr-x 3 yunhao yunhao 4096 cmake
drwxrwxr-x 2 yunhao yunhao 4096 coverage
drwxrwxr-x 2 yunhao yunhao 20480 db
drwxrwxr-x 14 yunhao yunhao 4096 docs
drwxrwxr-x 2 yunhao yunhao 4096 env
drwxrwxr-x 2 yunhao yunhao 4096 examples
drwxrwxr-x 2 yunhao yunhao 4096 hdfs
drwxrwxr-x 3 yunhao yunhao 4096 include
drwxrwxr-x 7 yunhao yunhao 4096 java
drwxrwxr-x 2 yunhao yunhao 4096 memtable
drwxrwxr-x 2 yunhao yunhao 4096 monitoring
drwxrwxr-x 2 yunhao yunhao 4096 options
drwxrwxr-x 2 yunhao yunhao 4096 pmdk
drwxrwxr-x 3 yunhao yunhao 4096 port
drwxrwxr-x 2 yunhao yunhao 12288 table
drwxrwxr-x 2 yunhao yunhao 4096 test_sh
drwxrwxr-x 4 yunhao yunhao 4096 third-party
drwxrwxr-x 5 yunhao yunhao 4096 tools
drwxrwxr-x 2 yunhao yunhao 12288 util
drwxrwxr-x 27 yunhao yunhao 4096 utilities
```

图 4. 项目代码结构

本项目中的代码结构比较复杂，如4所示，有很多个文件夹，此处主要介绍起主要作用的文件夹，具体如表1所示。

表 1. 源代码作用表

目录名	作用
build_tools	编译过程中需要的一些工具
cache	数据库缓存管理
cmake	项目编译过程的 cmake 信息
db	详细的数据库插入，查询等操作的实现
env	环境控制
include	相关的.h 文件
memtable	memtable 的实现
monitoring	数据库运行状态监控
options	数据库的一些选项设置
pmdk	持久内存管理套件
table	数据库中表的管理
util	一些测试等的外部工具

MatrixKV 是基于 Rocksdb 构建的。我们修改了几个文件（即 db_impl.cc, version_set.cc、memtable.cc、skiplist.h 和 arena.cc），以支持使用字节寻址非易失性内存的 LSM 树的 L_0 层级。大约有 1,000 行代码被添加到原始的 Rocksdb 中。

4.2 实验环境

我们在具有 NVM 和 SSD 混合存储的系统之上实现本实验。采用物理服务器进行实验，服务器集成了带有 72M 缓存的 Intel Xeon Gold 5318 处理器，大小为 64G 的 DRAM，大小为 128GB 的 Optane 内存和大小为 2T 的 SSD。我们采用 Intel 持久内存开发套件（PMDK）来简化对 NVM 设备的管理和访问，使用直接内存访问（DAX）和 mmap 功能。操作系统是 Ubuntu 22.04，Linux 内核版本为 5.19.0。

4.3 创新点

考虑到持久内存与传统 SSD 相比，具有的高读写带宽，而与 DRAM 相比，具有持久性。而本实验中 128G 大小的持久内存只在 L_0 层使用了极小一部分，存在持久内存容量的浪费现象，为了更充分的使用吃持久内存，我们在原 LSM 树在磁盘上的每一层之间插入了与上一层大小相同的持久内存层，这样可以更充分的利用持久内存的空间，并且也能一定程度上降低磁盘带宽的占用，提升系统整体的性能。

5 实验结果分析

我们在上文的环境中部署实验，并采用 YCSB（Yahoo! Cloud Serving Benchmark）来对整体性能进行测试。YCSB 是一种开源规范和程序套件，用于评估计算机程序的检索和维护

能力。它通常用于比较 NoSQL 数据库管理系统的相对性能。

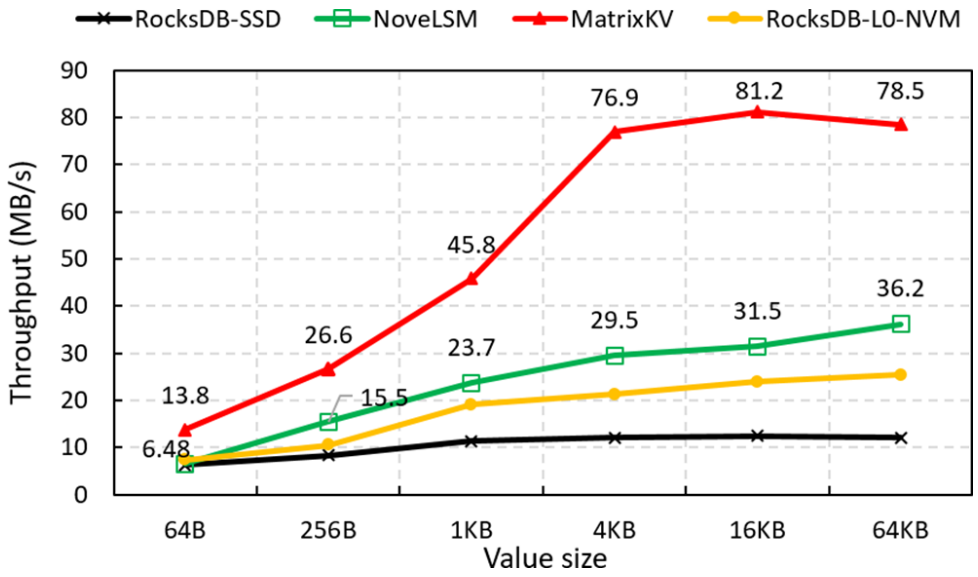

图 5. 随机写带宽

我们通过插入总计 80GB 的 KV 项来评估随机写性能，这些 KV 项以均匀分布的随机顺序插入。图5显示了四个 KV 存储在不同值大小下的随机写吞吐量。RocksDB-SSD 与 RocksDB-L0-NVM 之间的性能差异表明，仅将 L_0 放置在 NVM 中就能带来平均 65% 的改进。我们使用 RocksDB-L0-NVM 和 NoveLSM 作为评估的基线。MatrixKV 在所有值大小上都提高了相比于 RocksDB-L0-NVM 和 NoveLSM 的随机写吞吐量。具体来说，MatrixKV 相比于 RocksDBL0-NVM 的吞吐量提升范围从 1.86× 到 3.61×，相比于 NoveLSM 的吞吐量提升范围从 1.72× 到 2.61×。以常用的 4KB 值大小为例，MatrixKV 分别比 RocksDB-L0-NVM 和 NoveLSM 高出 3.6 倍和 2.6 倍。RocksDB-L0-NVM 的性能相对较差，因为将 L_0 放置在 NVM 中只带来了边际改善。NoveLSM 在 NVM 中使用了一个大的可变 MemTable 来处理一部分更新请求，从而稍微减少了写放大（WA）。然而，对于 RocksDB 和 NoveLSM 来说，写停顿和 WA 的根本原因仍未解决，即全对全的 L0-L1 压缩和 LSM 树的加深层次。

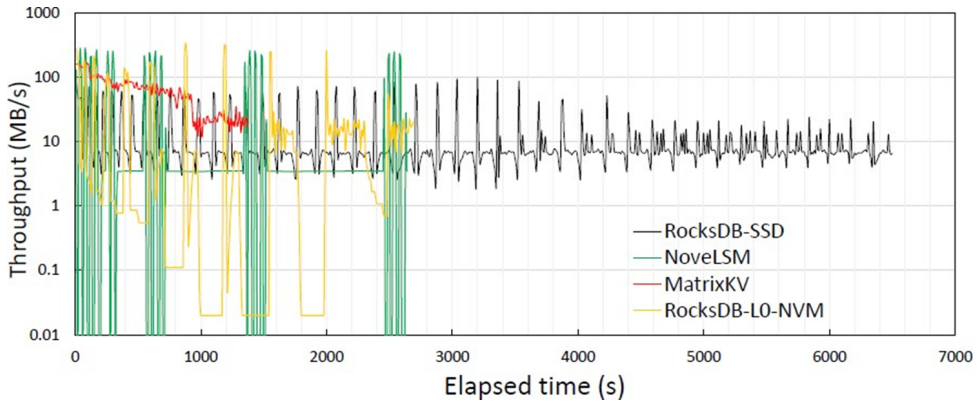

图 6. 一定时间内带宽的波动

我们记录了这四个 KV 存储在 80GB 随机写过程中每十秒的吞吐量，以可视化写停顿。从图6显示的性能变化中，我们得出三个观察结果。MatrixKV 在处理相同的 80GB 随机写时所

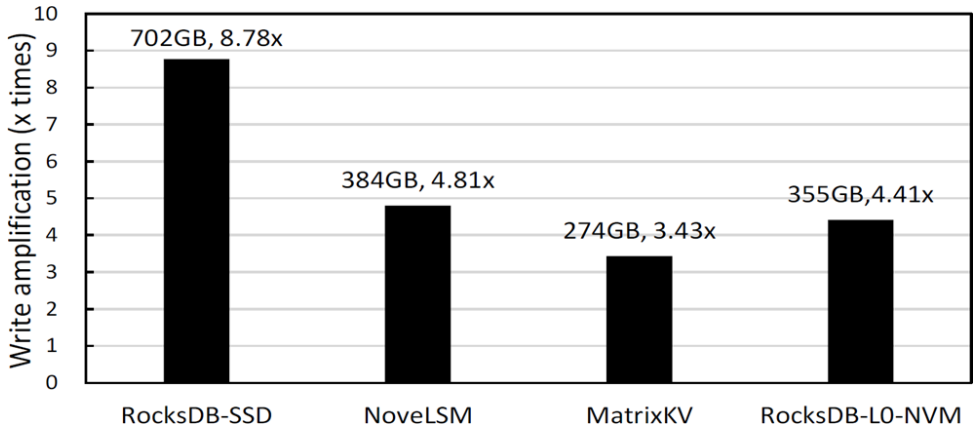

图 7. 80GB 随机写入的写放大

需时间更短，因为它的随机写吞吐量高于其他 KV 存储。由于昂贵的 $L_0 - L_1$ 压缩，RocksDB 和 NoveLSM 都遭受写停顿。NoveLSM 处理 $L_0 - L_1$ 压缩所需时间更长，因为 L_0 维护了从 NVM 刷新的大型 MemTables。与 RocksDB-SSD 相比，RocksDB-L0-NVM 在写停顿期间的吞吐量更低，这意味着由于 L_0 扩大，它更严重地阻塞了前台请求。MatrixKV 实现了最稳定的性能。原因是我们通过细粒度的列压缩减少了写停顿，它保证了在每次 $L_0 - L_1$ 压缩中处理的数据量很小。

我们在相同的实验中，即随机写入 80GB 数据集，测量了四个系统的写放大（WA）情况。图7显示了通过 SSD 写入的数据量与用户数据量之比来衡量的 WA 因子。MatrixKV、NoveLSM 和 RocksDB-L0-NVM 的 WA 分别比 RocksDB-SSD 低 2.56 倍、1.83 倍和 1.99 倍。MatrixKV 的 WA 最小，因为它通过降低 LSM 树的深度，减少了压缩的次数。

6 总结与展望

在本文中，我们介绍了 MatrixKV，一种基于 LSM 树的稳定低延迟键值存储系统。MatrixKV 为具有多层 DRAM-NVM-SSD 存储的系统而设计。通过将 L_0 层提升到 NVM，使用矩阵容器管理它，并通过细粒度的列压缩来压缩 L_0 和 L_1 层，MatrixKV 减少了写停顿。通过展平 LSM 树，MatrixKV 减轻了写放大。MatrixKV 还通过跨行提示搜索保证了足够的读取性能。MatrixKV 基于 RocksDB 在实际系统上实现。评估结果表明，MatrixKV 显著减少了写停顿，并且比 RocksDB 和 NoveLSM 实现了更好的系统性能。

这次的复现工作使我获益良多，让我对 LSM 树的存储结构有了进一步的认识，并且首次尝试了对顶会论文进行改进，这是一次很宝贵的体验。从上述的实验结果分析可见，引入 NVM 并优化 L_0 层结构，可以得到更大的带宽和更低的写停顿以及写放大。但是是否可以通过其他优化算法，或其他更恰当的结构来进一步提升 LSM 树的性能则有待考量。如果在 LSM 树的优化过程中，结合混合存储介质，选择合适的优化算法来提升系统整体性能仍是个值得探讨的问题。

参考文献

[1] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, Oct 2009.

[2] RemziH. Arpaci-Dusseau and AndreaC. Arpaci-Dusseau. Operating systems: Three easy pieces. Feb 2015.

[3] Joy Arulraj and Andrew Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, May 2017.

[4] Anurag Awasthi, A. Nandini, Arnab Bhattacharya, and Priya Sehgal. Hybrid hbase: leveraging flash ssds to improve cost per throughput of hbase. *International Conference on Management of Data,International Conference on Management of Data*, Dec 2012.

[5] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. Triad: creating synergies between memory, disk and log in log structured key-value stores. *USENIX Annual Technical Conference,USENIX Annual Technical Conference*, Jul 2017.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. Silk: Preventing latency spikes in log-structured merge key-value stores. *USENIX Annual Technical Conference,USENIX Annual Technical Conference*, Jan 2019.

[7] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. Flodb. In *Proceedings of the Twelfth European Conference on Computer Systems*, Apr 2017.

[8] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. Bankshot. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, Nov 2013.

[9] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4.5):449–464, Jul 2008.

[10] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey. In *Proceedings of the 2017 ACM International Conference on Management of Data*, May 2017.

[11] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *Proceedings of the 2019 International Conference on Management of Data*, Jun 2019.

[12] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. In *Proceedings of the Thirteenth EuroSys Conference*, Apr 2018.

[13] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. *File and Storage Technologies,File and Storage Technologies*, Jan 2018.

[14] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, AmirSaman Memaripour, YunJoon Soh, Zixuan Wang, Yi Xu, SubramanyaR. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane dc persistent memory module. *arXiv: Distributed, Parallel, and Cluster Computing,arXiv: Distributed, Parallel, and Cluster Computing*, Mar 2019.

[15] Olzhas Kaiyrakhmet, SongYi Lee, Beomseok Nam, SamH. Noh, and Young-ri Choi. Slm-db: Single-level key-value store with persistent memory. *File and Storage Technologies,File and Storage Technologies*, Feb 2019.

[16] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, AndreaC. Arpaci-Dusseau, and RemziH. Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. *USENIX Annual Technical Conference,USENIX Annual Technical Conference*, Jul 2018.

[17] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. *File and Storage Technologies,File and Storage Technologies*, Feb 2014.

[18] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, Oct 2019.

[19] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage*, page 1–28, Feb 2017.

[20] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb. In *Proceedings of the 26th Symposium on Operating Systems Principles*, Oct 2017.

[21] Russell Sears and Raghu Ramakrishnan. blsm. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, May 2012.

[22] Pradeep Shetty, RichardP. Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. *File and Storage Technologies,File and Storage Technologies*, Feb 2013.

[23] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, page 80–83, May 2008.

[24] Xingbo Wu, Yuehai Xu, Zehui Shao, and Song Jiang. Lsm-trie: an lsm-tree-based ultra-large key-value store for small data. *USENIX Annual Technical Conference,USENIX Annual Technical Conference*, Jul 2015.