分　数：	
评卷人：	

華中科技大學

研究生数据中心技术课程报告

题 目：基于混合DRAM-NVM的高性能键值存储方案

学　　号　M202173483

姓　　名　孙依茗

专　　业　电子信息

课程指导教师　施展 童薇

院（系、所）　武汉光电国家研究中心

基于混合 DRAM-NVM 的高性能键值存储方案

孙依茗 [1)]

[1)](华中科技大学武汉光电国家研究中心 湖北 武汉 430074)

摘 要 持久性键值存储是数据中心存储基础设施的重要组成部分，在现代的软件系统中也得到了广泛的应用。新兴的非易失性存储器 NVM 以其接近 DRAM 的速度和接近 SSD 的容量，为未来存储器架构设计提供了新的选择。然而简单地用 NVM 替换现有的存储并不能实现高效的键值存储，因为基于块访问 NVM 和磁盘表现出的行为并不相同，并且将会忽略 NVM 的字节可寻址性、布局和独特的性能特征。本文介绍了三种基于新型存储器 NVM 和 DRAM 混合存储的高效的键值存储，分别叫做 HBTree、RangeKV 以及 Viper。HBTree 提出了一种新型的基于 DRAM-NVM 混合的键值存储索引结构，利用实际应用场景中数据访问的冷热特性，将热数据缓存到 DRAM 中，从而提高 KV 运行效率；同时，应用日志记录功能以保证对缓存数据进行写操作时的数据可靠性。RangeKV 为运行在基于 NVM 的混合内存系统中的应用程序提供低延迟和高吞吐量。RangeKV 则是基于 lsm 树的一种异构存储架构的持久性键值存储设计。RangeKV 使用 NVM 中的 RangeTab 来管理 L0 数据，通过预先构造 RangeTab 数据的哈希索引来减少 NVM 的访问次数，并采用双缓冲区结构来减少由于压缩导致的 LSM-tree 写放大。Viper 的设计中提出了三种特定于 pmems 的访问模式，并在一个混合 NVM-DRAM 的架构中实现。Viper 采用基于 DRAM 的哈希索引和 pmems 感知存储布局，利用 DRAM 的随机写入速度和 PMem 的高效顺序写入性能。

关键词 键值存储；NVM；混合存储；

中图法分类号 TP302

High Performance Key-Value Store scheme based on nonvolatile memory

Sun Yiming[1)]

[1)](Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei)

Abstract Persistent key-value storage is an important part of data center storage infrastructure, and it has also been widely used in modern software systems. The emerging non-volatile memory NVM, with its speed close to DRAM and capacity close to SSD, provides a new choice for future memory architecture design. However, simply replacing existing storage with NVM cannot achieve efficient key-value storage, because block-based access NVM and disks exhibit different behaviors, and the byte addressability, layout and uniqueness of NVM will be ignored. Performance characteristics. This article introduces three high-efficiency key-value storage based on the new memory NVM and DRAM hybrid storage, which are called HBTree, RangeKV and Viper. HBTree proposes a new type of DRAM-NVM hybrid key-value storage index structure, which uses the hot and cold characteristics of data access in actual application scenarios to cache hot data in DRAM, thereby improving the efficiency of KV operation; at the same time, application log records Function to ensure data reliability when writing cached data. RangeKV provides low latency and high throughput for applications running in NVM-based hybrid memory systems. RangeKV is a persistent key-value storage design of a heterogeneous storage architecture based on the lsm tree. RangeKV uses RangeTab in NVM to manage L0 data, reduces NVM access times by pre-constructing a hash index of RangeTab data, and uses a double buffer structure to reduce LSM-tree write amplification caused by compression. In Viper's design, three pmems-specific access modes are proposed

and implemented in a hybrid NVM-DRAM architecture. Viper uses a DRAM-based hash index and pmems-aware storage layout, using the random write speed of DRAM and the efficient sequential write performance of PMem.

Key words Key-Value store; NVM; hybrid storage

1 引言

持久性键值存储(KVS)已经成为除了传统关系数据库管理系统(RDBMS)之外的一种广泛使用的数据存储类型。与 RDBMS 不同，键值存储可通过给定键检索无模式的数据。键值存储的工作负载也不同于经典的 RDBMS 工作负载，因为它们写入量较大，而且几乎只对单个记录[1]进行操作。根据使用的索引结构，键值存储可以分为三类：基于哈希索引的设计基于 B 树的设计和基于日志结构合并(LSM)树的设计。

为了确保数据持久性，目前的键值存储将数据写入具有基于块接口的设备，即 SSD 或 HDD。然而，持久性内存(又称 PMem、NVRAM 或 NVM)的出现，提供了接近 DRAM 的读写速度、可字节寻址的能力和数据持久性。因此，在键值存储中使用 NVM，在提高键值存储的性能方面有很大的潜力。

为了提高写量大的工作负载的性能，大多数传统的持久 KVM(如 RocksDB[2]或 LevelDB[3])都会优化它们的插入操作，以避免在基于块的设备上进行昂贵的写放大。因此使用日志结构的树[4]来收集内存中的记录，然后将这些记录以单个块大小的块写入磁盘。这种方法需要额外的基于磁盘的提前写日志，以确保数据持久性，以及用于磁盘写的复杂的合并逻辑。此外，大多数基于磁盘的 KVM 需要字符串或字节键和值来存储任意数据。这使得每次访问的反序列化成本很高，严重影响了整体性能[5][6]。不论是基于哈希索引的设计、基于 B 树的设计和基于日志结构合并树的设计都存在着一定的问题，也并不能将 NVM 的性能优势发挥到最大。因此目前的一些研究分别从这三种不同的方面对基于 DRAM 和 NVM 的键值存储进行优化

为了克服这些问题，Lawrence Benson 等人提出了一种基于哈希索引的键值存储优化方案 Viper。他们提出了三种特定于 NVM 的访问模式，以便在混合 NVM-DRAM 环境中高效地直接存储和检索 NVM 中的数据。他们采用基于 DRAM 的哈希索引和 NVM 感知存储布局，同时利用 DRAM 的随机写入速度快和 NVM 的高效顺序写入性能；将数据直接存储在 NVM 中，将 Viper 与最先进的 KVS 进行的评估中表明它在核心 KVS 操作方面均有优势。在写工作负载方面，Viper 比现有的仅使用 NVM、混合存储和基于磁盘键值存储的性能高出 4-18 倍。

LSM 树是一种针对高插入数据(如事务性日志数据)而提出的数据结构。LSM 树以两个或多个独立的结构维护数据，每个结构都针对其各自的底层存储介质进行了优化。在两个结构之间高效地批量同步数据。实践中使用的 LSM 树的大多数实现都使用多个级别。级别 0 被称为 Memtable，它保存在主内存中，可以用树来表示。磁盘上的数据被组织成有序运行的数据，这些数据被称为 sstable。每个 SSTable 包含按索引键排序的数据。

LING ZHAN 等人提出的 RangeKV 是一种基于 LSM 树的基于异构存储架构的持久键值存储；这项设计中使用 NVM 中的多个 RangeTab 结构来组织 LSM 树中的 L0 层，以缩短压缩延迟。RangeTab 以块的形式存储从 DRAM 中刷新的数据作为基本单元。此外，在 LSM-tree 的压缩过程中采用了双缓冲结构，减少了系统写阻塞和写放大的延迟。研究者基于 RocksDB 实现了 RangeKV 的设计，并与 RocksDB 和 NoveLSM 进行了对比测试和性能评估。测试结果表明，RangeKV 的随机写性能是 RocksDB 的 4.5 ~ 5.7 倍，压缩次数比 RocksDB 低 50%以上，平均合并数据量降低了约 40%。该系统的写入放大只有 RocksDB 的 25%左右。此外，与 NoveLSM 解决方案相比，RangeKV 具有显著的性能优势。

在基于 B 树设计的键值存储方面，Yuanhui Zhou 等人提出了混合 B+树的 HBTree 的索引结构，提高索引结构的性能，缩短重构时间。索引是持久键值存储的一项基本技术。索引结构的运行效率在很大程度上决定了 Put、Get、Delete 等键值操作的效率，是存储领域的研究热点之一。传统的索引结构不适合基于 NVM 或 DRAM-NVM 混合存储系统，因为它们是专门为硬盘或者 SSD 设计的。研究者提出的 HBTree 基于实际应用场景中数据访问的冷热特性。

通过将热数据所在的 NvmTree 放置在 DRAM 中，通过日志的方式保证了 DRAM 上 Cachetree 的一致性，从而提高热数据的读写性能。同时，对 DRAM 上的索引进行备份，减少系统故障恢复时间。

2 原理和优势

2.1 非易失性存储器

与传统的磁盘相比，Optane DC PMM 等新兴的非易失性存储器(NVM)是一种新兴的存储设备，弥合了 DRAM 和基于闪存的存储之间的差距。它结合了 DRAM 提供的字节可寻址数据访问和二级存储的持久性，同时提供接近 DRAM 的速度。英特尔最近公开了其 Optane DC 持久内存内存[7]。这些内存采用 3D XPoint 技术，比 DRAM 内存密度高，容量大，每 GB 成本低。当然，NVM 也存在容错性差、读写性能不对称等缺点。由于 NVM 的持久性特点，为了保证系统故障恢复的正确性，必须考虑 NVM 中持久性数据的一致性。

2.2 键值存储

键值存储（KVS）是一类存储系统，它以<key、value>对处理数据。键值存储实现的基本操作包括 put、get、delete 和可选的 update[2][8][9]。访问 KVS 有两种设计，分别是 KVS 服务器和嵌入式 KVS。基于服务器的 KVS 存储和同步状态可以被运行在不同机器上的多个应用程序全局访问。它通过网络客户机/服务器 API 与应用程序通信。当前较为流行的 KVS 服务器是 *Redis*[10]和 *memchached*[11]。如果 KVS 被单个应用程序使用，嵌入式 KVS 比基于服务器的 KVS 更轻量级。流行的嵌入式 KVS 是 RocksDB[2]和 FASTER [8]。KVS 服务器的一个主要优点是它们是自包含的系统。这为它们提供了完全的系统控制，例如，它们管理自己的线程、并发性和 I/O 队列。然而，这种控制需要抽象成本，例如，基于网络的接口。另一方面，嵌入式 kvm 是由用户在应用程序中控制的，与基于网络的访问相比，这导致了更少的通信开销，并允许更多的微调。但是，这种控制存在不正确使用的风险，可能会影响正确性和性能。为了提供良好的性能和控制，嵌入式 KVS 必须将其接口设计得尽可能简单，而不要求用户在出现部分故障或系统重启等情况时严格遵循模式或复杂的过程。

2.3 一种基于NVM的键值存储NovelSM

NoveLSM 开发两个方案使用 NVM 和 SSD 作为混合存储结构:在方案 1 中,NVM 存储不可变数据,如图 1（a）所示；方案 2 则是当 DRAM 中的 memtable 数据几乎满时，将数据写入 NVM 的 memtable 中，如图 1 (b)所示。为了更好地解决 memtable 写入 NVM 时系统处理写请求时阻塞的问题，NovelSM 最后选择了方案 2。

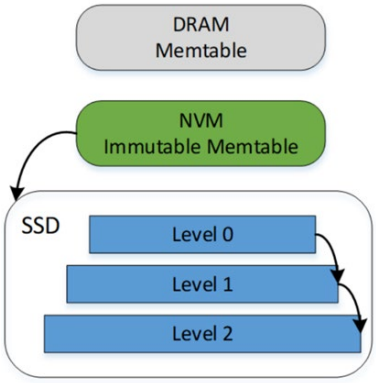

图 1（a） NovelSM 方案 1

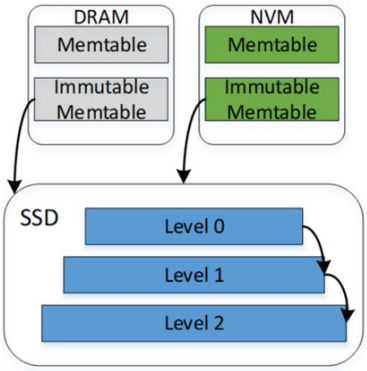

图 1（b） NovelSM 方案 2

但是研究者通过随机写入相同的 80 GB 数据库来评估使用 8 GB NVM 的 NovelSM。通过分析图 2 所示的测试结果，研究者发现以下问题：

1) NoveLSM 中的 NVM memTable 具有更高的容量。当它已满时，需要将数据刷新到 SSD。但数据量（4GB）较大，系统性能明显下降。在图 2 中红色椭圆覆盖区域对应的时间间隔内，可以观察到系统性能出现频繁的波动。

2) 当 L0 级 SSTable 的数量过多时，会触发压缩。由于合并数据量较大，刷新线程阻塞等待时间延长，对应图 2 中黑色椭圆覆盖的曲线对应的时间间隔。

3) 系统写入的数据量越大，压缩次数和 LSM 树级别之间的数据量就越高。 压缩线程占用

的时间越长，刷新线程阻塞的等待时间就越长，但系统的写性能处于最低水平的时间越长。因此，图 2 中黑色椭圆覆盖区域所指示的时间范围也在逐渐扩大。

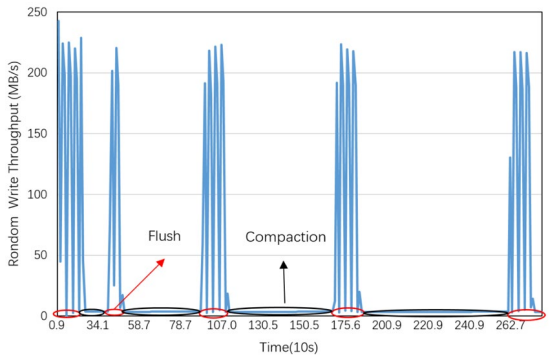

图 2 NoveLSM 的随机写入性能（每 10 秒测量一次）

但是 NoveLSM 没有为 LSM 树的压缩操作提供具体的优化措施。系统中的主要瓶颈仍然是向 SSD 写入数据的性能。

2.4 基于B+树的索引优化方案对比

在运行效率方面，FAST&FAIR 和 FPTree 是两种典型的基于 NVM 的 B+树索引结构的优化方案。图 3 对这两种解决方案的运行效率进行了评价，并与 DRAM 上的 B+树和 DRAM 解决方案上带日志的 B+树(DRAM-B+treeLog)的运行效率进行了比较。其中，DRAM 上的 B+树利用了广泛使用的 stx-btree[12]。

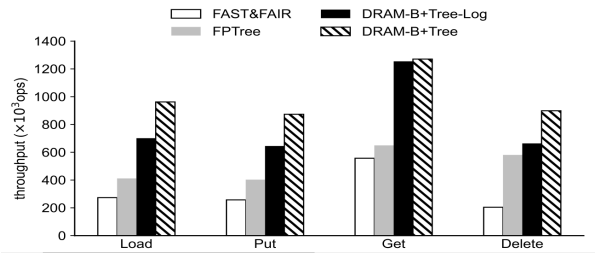

图 3 四种索引结构的吞吐量(Kops/s)

基于 DRAM-NVM 混合结构索引的 FPTree 的性能优于仅基于 NVM 的 FPTree，但低于 DRAM 上的 B+树解决方案。与基于 NVM 的 B+树相比，DRAM-B+tree-Log 有一个明显的弱点，因为系统重启时需要重放所有的日志数据。在这种情况下，该解决方案耗时较长，并且在数据量较大时会占用过多的 DRAM 资源。

3 研究进展

3.1 基于哈希索引的键值存储优化

Viper 由位于 DRAM 中的易失哈希索引和位于 NVM 中的持久数据块组成。研究者提出了三种高效访问 NVM 的方式，分别为**直接写入 NVM**；由于顺序 PMem 写入速度比之前模拟中假设的要快，Viper 将所有数据直接写入 NVM，而不需要中间的 DRAM 缓冲区。**均匀的线程到内存分布**；Viper 通过将线程分配给不同的内存区域，使插入的线程与内存的比率最小化。**内存对齐存储段**；Viper 将数据存储在 DIMM 边界对齐的 VPages 中，以平衡内存竞争与并行性。

图 4 所示为使用不同数量的线程来执行 64 字节的存储测试来对比 NVM 和 DRAM 的性能差异，从图 4（a）中可以看出顺序写的方式中，NVM 的性能与 DRAM 十分相近，这是由于 NVM 内部缓冲区进行合并操作，从而减少了刷新操作。图 4（b）中则可以看出随着线程数的增多，随机写的延迟增高的很快，NVM 内部的合并操作并不能合并小的随机写操作，因此研究者在设计中将选择顺序的直接写入而尽量避免随机写入。

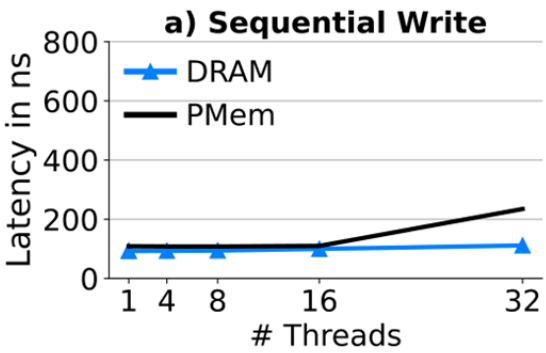

图 4（a）顺序写延迟对比

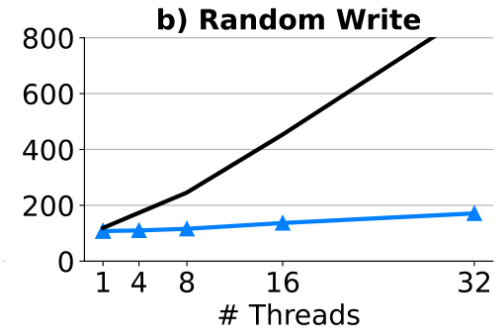

图 4（b）随机写延迟对比

图 4（c）显示了第三个设计在所有内存中均匀分布线程的重要性。研究者将线程分配到 k 个内存区域(每个内存区域 1GB)，并顺序地写入这些文件。使用一个日志文件(在图中表示为 1)，所有线程都写入相邻的缓存行，即线程 1 写入字节 0-63，线程 2 写入字节 64-127，以此类推。当使用相同数量的线程和日志时，每个线程都有自己的分离内存区域。使用更多的日志，更少的线程共享内存区域，并均匀地分布在内存中。一个线程的性能较差是由于所有线程都运行在一个内存上，忽略了 NVM 固有的并行性。

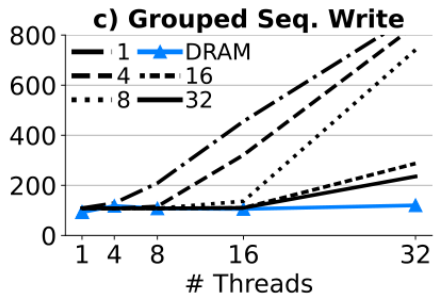

图 4（c） NVM 不同线程下的延迟

图 5 所示为 Viper 的设计结构，Viper 主要是基于 DRAM 和 PMem 混合设计的，因此在易失的 DRAM 用于存储哈希索引，也就是图 5 中左侧所示存储每个记录的键和它在 PMem 中的存储位置。然后我们将键值记录存储在持久存储的 Vpage 中，几个 VPgae 组成一个 VBlock。

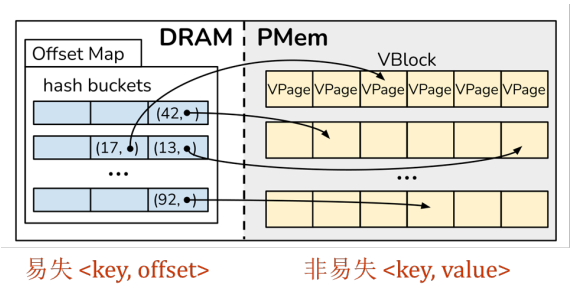

图 5 Viper 结构设计

VPgae 的结构如图 6 所示，可以看到一个 VPage 的大小是 4KB，在 VPage 的首部有一些的元数据，包括用于处理并发访问的锁位和用于指示本页中哪些 slot 可用哪些不可用的位集合。后边是许多的 slots，用于存储键值记录。在这里我们 VPage 的设计也体现了我们前边所述的设计选择，首先我们 VPage 的大小设置为 4KB，也即研究者所提出的页对齐的存储段；同时因为写入顺序是按照 slot 顺序写入的，因此也满足之前所提到的使用顺序写入 PMem，避免随机写的设计。

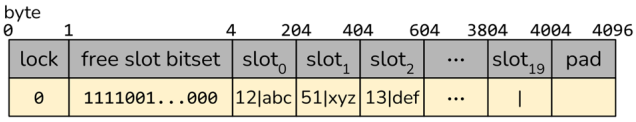

图 6 VPage 结构设计

研究者第三个设计理念的体现在为每个不同区域的 DIMM 分配不同的线程。因此研究者引入了 Viper 客户端，这个客户端并不是像服务器客户端那样含有网络的逻辑，而是一个轻量级的对象，是一个用户链接数据库，并且通过指令进行操作、存储一些元数据的接口。如图 7 中所示，假设现在有三个客户，这些客户将被 viper 分配他们自己的 VBlock，每个客户都能够顺序的写入自己 VBlock 中的 VPage。当一个 VBlock 存满了，客户段将向 viper 请求一个新的 VBlock 同时更新客户端相应的元数据。接下来客户将在新的 VBlock 中执行存储。这种方法减少了 Viper 内部的协调开销，因为不需要为每次写入数据查找新的位置。这种方式确保了线程在不同的 DIMM 上分布，而不是一个中心同步进行的方式。

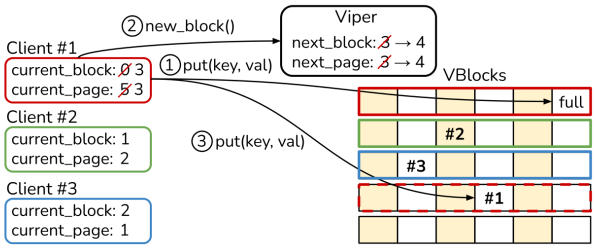

图 7 客户端请求新的 VBlock

接下来是 Viper 的测试结果。研究者首先在数据库中存放一亿条记录，然后再进行一亿次的其他操作，将 viper 与其他的一些基于 PMem 的键值存储，或者采用 DRAM 和 PMem 混合的键值存储进行对比。在图 8（a）中可以看到 Viper 在 put 操作中吞吐量表现的很好，主要是因为 Viper 将插入操作的随机 I/O 写入 DRAM 中的哈希索引中，再将记录顺序写入到 PMem 中，利用 DRAM 随机读写性能高和 PMem 顺序写性能高的优点。在图 8（b）中可以看出的 get 操作测试中可以看到 Viper 优于一部分的键值存储系统，这主要是因为无法避免对 PMem 的随机读取操作，但是 Viper 的 get 操作的总体性能优于其他几种。

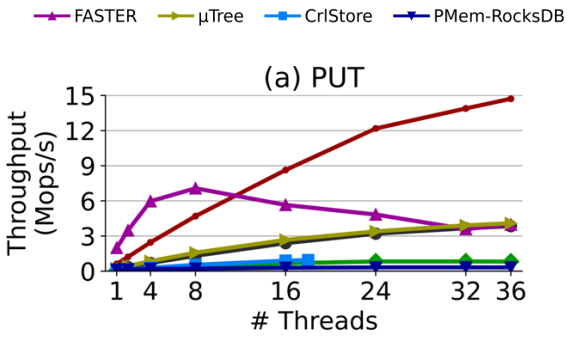

图 8（a） PUT 操作吞吐率对比

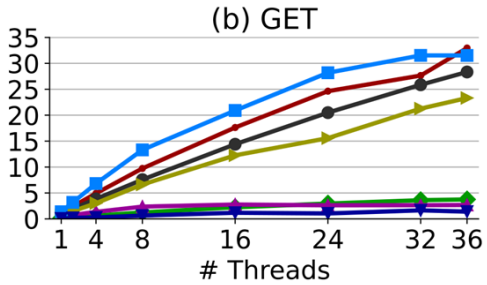

图 8（b） GET 操作吞吐率对比

在关于存储位置的测试中的测试的配置与之前的相同，但是这个实验旨在验证 viper 中数据存放的合理性。因此对比了将数据和索引全部存放在 DRAM 或者 PMem 中，将索引放在 DRAM，数据放在 PMem 中以及使用未对齐的 VPage 这几种情况。我们可以看到 Viper 比仅使用 PMem 的情况性能高出了三倍，这也是得益于 Viper 利用了 DRAM 随机访问的高性能。并且仅使用 DRAM 的测试组仅比 Viper 高了 1.6 倍。但是 Viper 比起来仅使用 DRAM 的存储系统又增加了持久性存储的部分。综上所述，Viper 能够高效的在 PMem 和 DRAM 混合存储中进行键值存储。

3.2 基于B+树索引的键值存储优化

HBTree 是一个混合的三层持久索引。如图 9 所示，HBTree 的数据结构包括索引层、中间层和数据层。

数据层是为了实现高效的扫描，利用 B+树来实现数据管理，但是由于数据量的增加导致 B+Tree 高度增加，需要更多的 NVM 访问。为了减少 NVM 访问和实现高效的索引操作，全局的 B+树被分解成许多小的 B+树，称为 NvmTree，键范围连续。另外，热 NvmTree 作为 CacheTree 缓存到 DRAM 中，并使用 Log 来保证一致性。

中间层是为了有效地管理数据层的 Log-Tree，设计了中间层，它是一个双链表，每个键代表一个 LogTree。中间层还会对每个 LogTree 的访问频率进行统计，并识别出存储在 DRAM 中缓存的热 NvmTree，从而提高读写速度。为了减少恢复时间，中间层的所有节点都会立即存储到 NVM 中。

索引层是 DRAM 中的一棵 B+树，用于索引中间层的元数据节点。它可以在中间层快速查找关键范围的元数据。通过遍历中间层，无需考虑系统断电或故障后的一致性，可以快速构建索引。

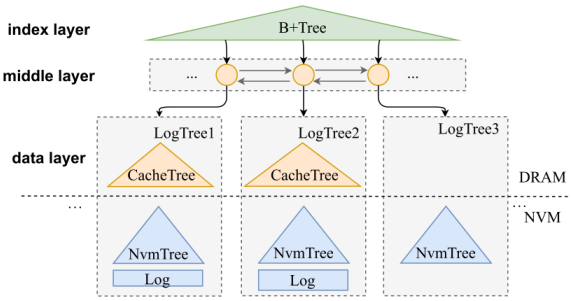

图 9 HBTree 整体结构

HBTree 可以看作是一个树结构，叶子节点是高度小、键范围连续的 LogTree。HBTree 有四个特点。首先，索引层全部位于 DRAM 中，不考虑一致性，提高了整体索引结构的效率。其次，DRAM 上的 CacheTree 和 NVM 上的 Log 的结合减少了整体结构的数据一致性开销。第三，对于大数据量，以 LogTree 为单位标识热数据和冷数据，只有热数据的 LogTree 在 DRAM 中才有 CacheTree，有效利用了少量的 DRAM 资源。最后，通过定期备份 CacheTree，可以减少系统掉电情况下的恢复时间。重构 CacheTree 后，可以快速重构数据较少的索引层和中间层，从而缩短索引结构的恢复时间。

中间层存储 LogTree 的元数据信息，包括 CacheTree 和 NvmTree 的根节点指针、LogTree 的 Log 记录和温度记录。当 LogTree 中的数据发生变化时，会立即更新 DRAM 上的 Cache 节点和 NVM 上的 persistent 节点。根据日志树的访问频率，可以识别出热日志树。因此，热日志树的 NvmTree 可以缓存到 DRAM 中，并定期进行同步，减少了 NVM 的访问，提高了性能。研究者根据 LogTree 的历史访问信息，提出了一种热统计算法来计算 LogTree 的温度。如果 LogTree 的历史访问频率较高，则在未来一段时间内，LogTree 可能会被频繁访问。

数据层中的 CacheTree 的写操作需要写日志，以保证数据的一致性。LogTree 的日志生成时，会分配日志空间，后续由较大的日志池回收。因为键和值是分开的，所以日志中只记录操作类型、键和

值指针。这样可以减少日志记录的大小，减少写开销。当写入日志时，首先添加日志记录，然后修改当前的分配地址。最后，通过 clflush 命令刷新日志记录和修改后的地址的缓存行，以确保数据持久性。对于被频繁访问的数据，在 LogTree 中有一个 CacheTree。写请求首先追加日志，然后将数据写入 CacheTree。这避免了直接写入较低的 NVM，并提高了写入性能。读操作可以直接查询 DRAM 中的 CacheTree。另外，为了减少大日志记录的恢复时间，CacheTree 会将脏节点同步更新回 NvmTree，然后回收日志。如果 LogTree 没有 CacheTree，则访问数据的频率较低，直接在 NVM 上访问 NvmTree 的延迟是可以接受的。NVM 保证写入数据的一致性，不写入日志。

研究者使用 YCSB 基准来评估 HBTree 索引的性能，并在该基准上执行 Put、Get、Update、Delete 和 Scan 操作。随机生成的扫描数小于 100，大多数 Workloads 使用默认配置。其中，Load C 使用 Latest 来生成一组热数据。所有工作负载都使用 8 字节和 8 字节的键值对来充分反映索引的性能。

在操作效率方面，将 HBTree 与两种典型的基于 pm 的索引结构 FAST&FAIR 和 FPTree 进行了比较。对均匀分布的 KV 数据进行了比较。YCSB 基准测试首先用 2 亿个键(称为 Load)填充索引，然后使用 1000 万个放和/或读请求运行相应的工作负载 A、B、C、D、E 和 F。工作负载 E 执行扫描操作，随机生成的扫描次数小于 100。所有方案的 B+树节点大小均为 512B。同时设置 HBTree 的缓存容量为 500MB。

实验结果如图 10 所示，可以看出，HBTree 在各种 YCSB 工作负载上的性能明显优于 FPTree 和 FAST&FAIR。工作负载 A 和 F 的结果表明，对于密集的写操作，HBTree 的性能提高相对较小。这是因为在 HBTree 写入 CacheTree 之前，还需要写入日志以保证数据的可靠性，并增加了一个持久化操作。

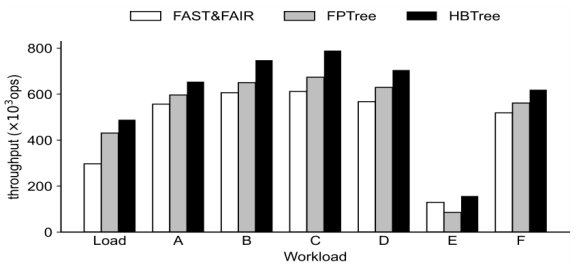

图 10 YCSB 工作负载下的吞吐量

为了探究 HBTree 的快速恢复能力，本节测试了 HBTree 在不同数据集大小和不同缓存容量下的故障恢复时间，并与 FPTree 进行比较。公平地说，HBTree 像 FPTree 一样使用单线程恢复。HBTree 的缓存大小设置为 500MB。从图 11 中可以看出，HBTree 的恢复时间更接近 FPTree。随着 key-value 对数量的增加，HBTree 的恢复时间保持在一个相对稳定的水平，而 FPTree 的恢复时间仍在增加。在此基础上实现了 FPTree 的恢复操作，时间开销主要集中在中间节点的重建上。HBTree 恢复的成本主要集中在 logtree 的恢复上。对于 200M 数据量的 key-value 对，HBTree 的恢复时间仅为 FPTree 的 30%。

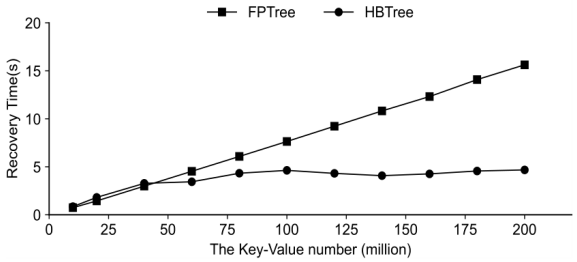

图 11 各种数据卷中 HBTree 和 FPTree 的恢复时间

综上所述，HBTree 索引结构可以充分利用有限的 DRAM 资源来提高系统性能，并利用 NVM 的特性来保证高效的数据持久性和快速恢复。

3.3 基于LSM树的键值存储优化

研究者基于 LSM 树的提出了基于三层异构存储系统的持久 KV 存储 RangeKV。图 12 显示了 RangeKV 的体系结构。RangeKV 按照存储结构分为三个级别:DRAM、NVM 和 SSD。在这个系统中，DRAM 层设计与 RocksDB 相同，使用 memtable 和 immutable memtable 批量写入请求；NVM 层包含 log 和 RangeTabs。日志保存了易失 DRAM 中数据的一致性和完整性。RangeTab 管理从 SSD 移动的未排序和重叠的 L0。通过新的压缩策略将数据合并到 L0 中；LSM-tree 中除 L0 外的其他各级存储在 ssd 盘中。

与 RocksDB 和 NoveLSM 相比，RangeKV 有三个设计目标。第一个是缩短压实延迟；RangeKV 采用 RangeTab 结构对 L0 中 KV 对进行重组，缩短压实延时。RangeKV 将多个 RangeTabs 映射到不同的和不重叠的键范围。memtable 数据根据键的范围被刷新到相应的 RangeTab。每次只有一个或几个相邻的 RangeTabs 参与压缩。这样可以缩短压缩实际涉及的关键范围，避免过多的数据在底层参与，从而

有效提高压缩效率，减少系统数据写阻塞的延迟。

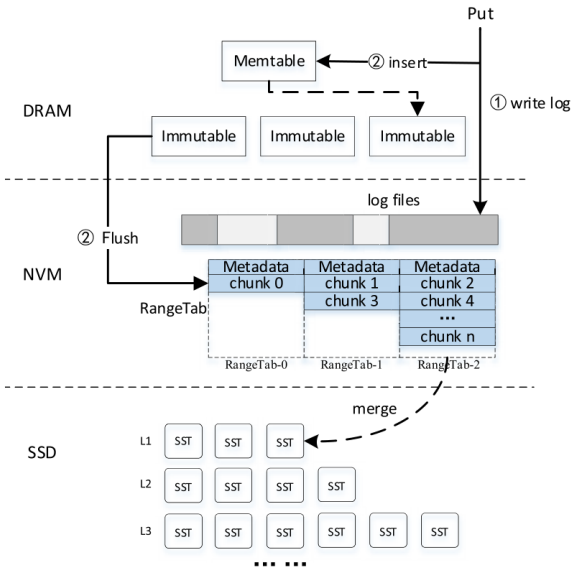

图 12 RangeKV 的系统架构

第二个目标是减少 LSM 树的级别。LSM-tree 级别越多，当系统写入相同数量的数据时，合并数据的次数就越多，这会增加写放大。特别是当系统写入数据量较大时，级别数对写性能的影响较大。RangeKV 通过适当增加 RangeTabs 的数量来增加所有 RangeTabs 的数据容量。在相邻层容量比例不变的情况下，RangeKV 可以在每层容纳更多数据的同时减少 LSM-tree 的层数。如图 13 所示，假设 RangeKV 共有 100 个 RangeTabs，每个 RangeTabs 的大小为 256MB，在其他因素不变的前提下，整个 L0 级的数据容量由 256MB 扩展到 25.6GB。当系统写相同数量的数据时，RangeKV 可以减少 LSM-tree 的实际级别，从而减少数据合并的次数，减轻压缩操作对系统写性能的影响

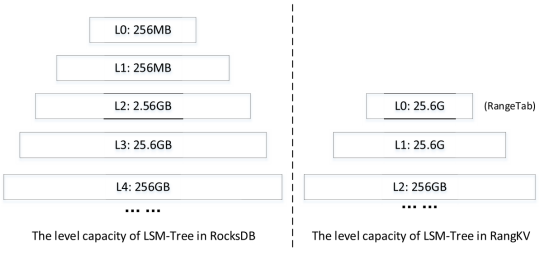

图 13 RangeKV 中减少 LSM 树级别

第三个优化目标是优化阅读性能；RocksDB 使用内存中的布隆过滤器来避免某些关卡中不必要的搜索。然而，布隆过滤器存在错误率的问题。搜索 L0 级别的数据时，关键值范围可能存在重叠。在使用二分法查找每个 SSTablefile 中的数据时，需要多次访问存储介质，这严重影响了系统在 L0 级查找数据的效率。RangeKV 通过存储在 NVM 中的 RangeTab 预先构造数据的哈希索引，以减少访问存储介质的次数。存储在 SSD 中的数据仍然组织在 SSTablefile 中，搜索方法也没有改变。RangeTab 的哈希索引方法可以显著减少 NVM 的访问次数，加快搜索速度，间接提高 RangeKV 系统的读性能。

研究者将 RangeKV 与 RocksDB-SSD、RocksDBNVM 和 NoveLSM 进行了比较。NoveLSM 是基于 LevelDB，而其他三款则是基于 RocksDB。为了进行公平的比较，所有四个数据库都使用一个线程进行压缩，一个线程进行刷新。memtables、immutable memtables 和 SSTable 的大小都是 64mb，这是 RocksDB 的默认配置。

在压实操作方面，RangeKV 的平均数据量仅为 225.2 MB，而 RocksDB 的平均数据量约为 370 MB，减少了约 40%。这主要是因为 RangeTab 的键范围和压缩策略限制了压缩键范围，从而减少了压缩过程中涉及的数据量。

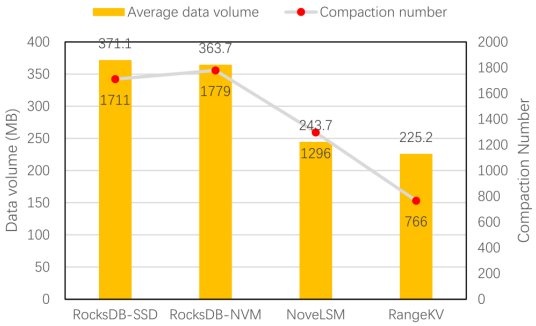

图 14 压实操作中的平均数据量

在减少 LSM 树级别方面，图 15 显示了不同 L1 尺寸下 RocksDB 和 RangeKV 的性能。测试结果表明，增大 L1 尺寸对 RocksDB 有负面影响，但对 RangeKV 性能有显著提高。对于 RocksDB 来说，将 L0 压缩到 L1 的开销会随着 L1 尺寸的增大而增大。然而，RangeKV 的压缩开销与级别大小无关，因为压缩粒度很细。

运行在 NVM 上的系统性能的测试结果如图 16 所示，可以看出 RangeKV 在三种系统中获得了最佳的随机写吞吐量。RangeKV 的优异性能证明了 RangeKV 比 NoveLSM 更好地利用了 NVM。在 DRAM-NVM 存储系统上，NoveLSM 的表现优于 RocksDB。

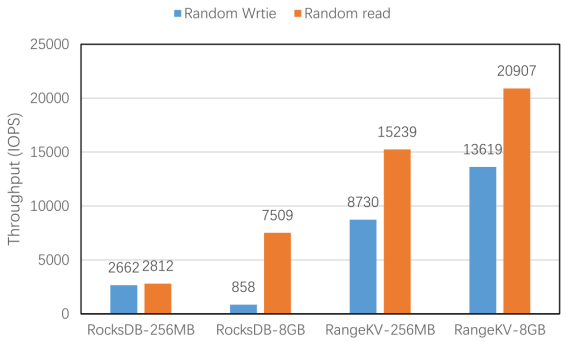

图 15 减低级别对比图

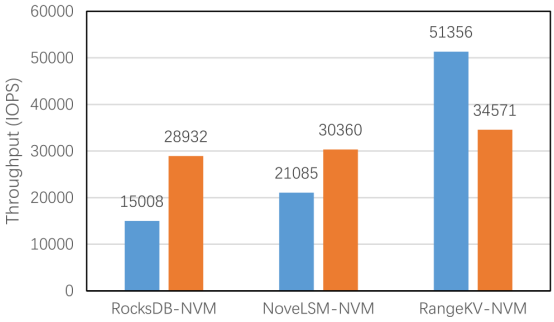

图 16 在 DRAM-NVM 中的系统性能

4 总结与展望

随着高性能的新型非易失性存储器 NVM 的出现，键值存储的架构等也将随之改变。如何利用 NVM 的特性设计出能够使其性能优势发挥最大的键值存储也成为了当今的一个热点研究问题，本文主要总结了三种基于不同索引结构的键值存储在 DRAM-NVM 混合的架构下的设计，三种设计考虑的问题出发点各不相同。Viper 是一种混合的 PMem-DRAM 键值存储，它利用 NVM 特定的访问模式来高效地存储和检索数据，同时提供完整的数据持久性。HBTree 索引结构则是充分利用有限的 DRAM 资源来提高系统性能，并利用 NVM 的特性来保证高效的数据持久性和快速恢复。RangeKV 是基于 lsm 树的持久 KV 存储，通过使用 RangeTab 将 L0 从 ssd 移动到 nvm，从而管理 L0，从而显示出较高的读写吞吐量。RangeKV 通过降低 LSM-tree 级别，使用双缓冲结构优化压缩，并预先构造哈希索引，可以减少写和读放大。基于 DRAM 和 NVM 混合存储的键值存储优化多种多样，Viper 的出发点为如何利用 DRAM 和 NVM 的特性，而 HBTree 和 RangeKV 则是从键值优化本身出发。在将来的研究中，这也是未来键值存储研究可以参考的研究方向。

参 考 文 献

[1] Lucas Lersch, Ivan Schreter, Ismail Oukid, Wolfgang Lehner. Enabling low tail latency on multicore key-value stores. Proceedings of the VLDB Endowment 13, 2020, 1091–1104

[2] Facebook. 2020. RocksDB. https://rocksdb.org.

[3] Google. 2020. LevelDB, a fast key-value storage library. https://code.google.com/p/leveldb.

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

[5] Apache Flink. 2020. Improvement in (de)serialization of keys and values for RocksDB state. https://issues.apache.org/jira/browse/FLINK-9702.

[6] Kazuaki Maeda. 2012. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In DICTAP'12. IEEE, 177–182.

[7] Intel. 2020. Intel® Optane™ Persistent Memory. https://intel.com/optanedcpersistentmemory.

[8] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In SIGMOD'18. ACM, 275–290.

[9] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. Proceedings of the VLDB Endowment 13, 7, 1091–1104.

[10] Redis. 2020. Redis, an in-memory data structure store. https://redis.io.

[11] Memchached. 2020. Memcached, high-performance, distributed memory object caching system. https://https://memcached.org/.

[12] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data structure primitives on persistent memory: an evaluation. In DaMoN'20. ACM, 14:1–14:3.

[13] Benson L, Makait H, Rabl T. Viper: an efficient hybrid PMem-DRAM key-value store[J]. Proceedings of the VLDB Endowment, 2021, 14(9): 1544-1556.

[14] Zhou Y, Sheng T, Wan J. HBTree: an Efficient Index Structure Based on Hybrid DRAM-NVM[C]//2021 IEEE 10th Non-Volatile Memory Systems and Applications Symposium (NVMSA). IEEE, 2021: 1-6.

[15] Lin Z, Kai L, Cheng Z, et al. RangeKV: An Efficient Key-Value Store Based on Hybrid DRAM-NVM-SSD Storage Structure[J]. IEEE Access, 2020, 8: 154518-154529.