

**研 究 生 ( 数 据 中 心 技 术 ) 课 程 论 文 ( 报 告 )**

**题目：基于持久性内存的KV存储优化研究**

|  |  |
| --- | --- |
| 学 号 | M202073165 |
| 姓 名 | 徐冰洁 |
| 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** |

2020年12月21日

基于持久性内存的KV存储优化研究

徐冰洁

1 引言

诸如PCM、ReRAM和3D XPoint等新兴的非易失性存储器（NVM）技术的兴起引起了学术界和工业界的广泛关注。NVM的一个潜在机会是充当位于存储器总线上的快速持久存储器，与传统硬盘或基于闪存的固态驱动器（SSD）相比，构建面向混合内存的KV存储这类的存储系统使人们能够利用快速内存​​访问来实现更高的性能。

持久键值存储（KV存储）已成为数据中心存储基础结构的重要组成部分。它们被广泛部署在大规模生产环境中，比如搜索引擎、电子商务平台和社交网络等等。在过去十年中，针对KV存储的设计和优化进行了大量研究，涉及诸如减少SSD的写入放大、减少索引带来的内存使用，并提高并发性以实现高可伸缩性等主题。常规的KV存储不适用于混合存储系统，因为它们是针对SSD的性能特征而设计的。例如，许多现有研究采用日志结构合并树作为索引结构，这避免了对硬盘或SSD的少量随机写入。与硬盘和SSD不同，混合存储系统是字节寻址的，并且为顺序访问和随机访问提供类似的性能。在为混合存储系统设计KV存储器时，保持大粒度的顺序写入会导致NVM的写入放大。

基于以上所述的器件和技术特性，本文简单地综述了近三年发表在计算机存储邻域优秀期刊上的三篇论文，他们提供了三种解决持久性内存中KV存储优化的有效方法。它们分别是基于日志结构的KV存储引擎、基于混合索引的KV存储引擎和一种写优化和高性能哈希索引方案，下面将分别介绍三种方案的实现方法。

2 问题与挑战

在持久性存储器中部署KV存储系统存在许多的问题和挑战，下面将从三个方面对其进行总结。

**确保一致性所产生的高开销**。发生系统故障时，持久性存储器中的数据结构应避免产生任何的数据不一致现象（即数据丢失或部分更新）。但是，通过内存总线直接访问NVM的新体系结构为了保证一致性会导致高性能开销。

首先，内存写入通常由CPU和内存控制器重新排序。为了确保一致性，必须采用高速缓存行刷新和内存防护，从而不可避免地引入了高性能开销。其次，现代处理器的原子写入单元通常不大于内存总线宽度（例如对于64位处理器来说原子写入单元为8字节）。如果写入的数据大小大于原子写入单元，则需要采用昂贵的日志记录或写时复制（CoW）机制来保证数据一致性。

**降低写入性能**。与读取相比，NVM中的内存写入消耗有限的耐久性，并导致更高的等待时间和能量。此外，对持久性内存的更多写入还会导致产生更多的缓存行刷新和内存隔离以及可能的日志记录或CoW操作，从而大大降低系统性能。因此，在NVM中减少写操作很重要。常见的哈希方案，例如链式哈希，跳房子哈希和布谷鸟哈希通常会导致许多额外的内存写入，从而才能处理哈希冲突，并且提出了诸如PFHT和路径散列之类的写友好散列方案，以减少散列索引结构中的NVM写操作，但其代价是降低了访问性能（即搜索、插入和删除操作的吞吐量）。

**调整哈希表大小的成本效率低下**。随着哈希表的负载因子（即存储项目数与总存储单元数之比）的增加，哈希冲突数增加，导致访问性能下降以及插入失败。调整大小对于哈希表在其负载因子达到阈值或发生插入失败时至关重要。

3 研究现状分析

3.1 基于日志结构的KV存储引擎

3.1.1 引言

KV存储已经成为数据中心架构的一个基本组成成分，因为其拥有简单抽象的接口（Put和Get等等）。最近，对快速数据存储和处理的需求不断地增长，从而引发了一个独特的问题，即如何设计有效的KV存储以处理当今生产部署中所必需面临到的写入密集型和小型工作负载。

新型的网络和存储技术（例如InfiniBand和PM）为实现这一目标带来了新的机遇：PCM、ReRAM和最近发布的Optane DCPMM，在提供数据持久性的同时还获得了与DRAM相当的性能和更高的密度。同样，最近发布的100 Gbps InfiniBand网络还提供了亚微秒级的延迟和极高的消息传递速率。本文的工作将研究如何借助此类新型设备最大化KV存储的性能，是因为我们对最近的许多KV存储深入分析后发现，这些存储的访问模式与PM中的访问粒度不匹配。

本文和前人的分析报告说，KV存储产生了大量的小型写入（例如，很大一部分KV对在生产工作负载中仅仅包含几个或十几个字节）。此外，KV存储中的索引结构导致写入的极大放大。对于基于Hash的索引，如果多个条目的key冲突或Hash表的大小已经调整，则需要重新散列多个条目。基于树的索引还需要经常移动每个树结点中的索引条目以使其保持有序，合并拆分树结点以保持树的平衡。大多数此类更新仅涉及更新一个或多个指针。CPU以cacheline的粒度（在AMD和x86平台上为64B）刷新数据，并且PM可能具有更粗糙的内部块大小（例如，在Optane DCPMM中为256B），在大多数情况下，该大小大于写入大写，大大浪费了硬件的带宽。

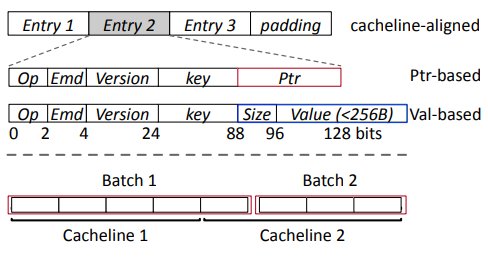
解决上述问题的经典方法就是使用日志结构来管理KV存储，因此可以将所有更新简单地附加到日志中。更重要的是，通过批处理技术处理来自客户端的多个请求并一起执行更新，可以在多个请求之间分摊写开销。但是，批处理的性能优势受到可以一起执行的更新数量的限制，这种想法对于HDD或SSD来说非常成功，因为他们具有更好的顺序写入性能，并且每个批次可以包含多个连续的扇区。但是在基于NVM的系统中的应用更具有挑战性：1）测量结果表明，只要I/O大小大于最小I/O单元（256B块大小），并且有足够的线程来执行I/O，对Optane DCPMM的顺序写入和随机写入具有非常相似的带宽，因此，批处理比单个I/O单元更多的数据是没有办法获益的。而且PM具有更精细的访问粒度（即64 B刷新大小和256 B块大小），如果我们仅在日志中记录每个内存更新，则PM仅能容纳非常有限数量的日志条目。2）批处理不可避免地增加了延迟，但是由于新兴的PM和高速NIC的主要优点之一就是低延迟，所以批处理技术在这些器件当中不能很好地被采用。由于这些挑战，我们发现，尽管最近的一些工作在基于PM的存储系统中采用了日志结构的思想，但它们主要采用日志结构来确保故障原子性或减少内存碎片，并未探索使用日志结构来增加利用存储空间的机会。

3.1.2 主要设计

1.紧凑的OpLog和Lazy-Persist分配器

在持久性KV存储中进行更新通常会导致写入和缓存刷新的巨大放大，因此为了解决此问题，FlatStore将KV存储解耦为易失性索引（通过直接使用现有的索引方案）以进行快速索引，压缩的每核OpLog来吸收较小的更新，以及用于存储大型KV的持久性分配器。要处理更新请求（即Put或Del），服务器核心只需写入KV项，在其本地日志的末尾追加日志条目，然后相应地更新易失性索引。因此，避免了由Hash索引中的重Hash和Tree索引中的移位、拆分和合并操作所引起的持久性开销。为了满足Get请求，服务器核心首先通过使用给定key引用的易失性索引来查找KV项，以找到确切的日志条目，最后通过KV项来定位。这样，也避免了扫描整个日志以查找特定key的开销。

**日志条目压缩**。OpLog的一个关键设计层面在于如何构造日志条目的布局：日志条目的设计应足够小以更好地支持批处理，从而使FlatStore能够在一次刷新操作中保留更多日志条目。 因此，FlatStore仅将索引元数据和超小型KV项放在OpLog中（在我们的实现中为256 B，足以饱和Optane DCPMM的带宽），而其他大KV项则分开存储在分配器中。同时，每个日志条目还需要包含足够的元数据以在运行时进行正常索引编制，并在系统崩溃后安全地恢复丢失的易失性索引结构。



我们结合操作日志技术来进一步压缩日志条目：日志条目仅包含用于描述每个操作的最少信息，而不是记录每个内存更新。如图所示，每个日志条目由五个部分组成，其中，Op记录操作类型（即Put或Delete），Emd表示是否将KV项放置在日志条目的末尾，引入了Version以保证日志清理的正确性，使用8字节key。 请注意，FlatStore可以将键放在OpLog之外，以支持更大的键，就像我们使用值一样。当Ptr存储在OpLog之外时，它用于指向实际记录。否则，该值及其大小将直接放在日志末尾。为了最小化日志条目的大小，我们仅为Ptr保留40位。这是可以接受的，因为分配器仅管理大小大于256字节的大型KV，因此忽略了低8位（40 + 8位指针能够索引128 TB NVM空间）。因此每个日志条目的大小限制为16个字节（用于基于指针的条目），指示可以一并清除16个日志条目，其开销等于持久保存单个日志条目的开销。

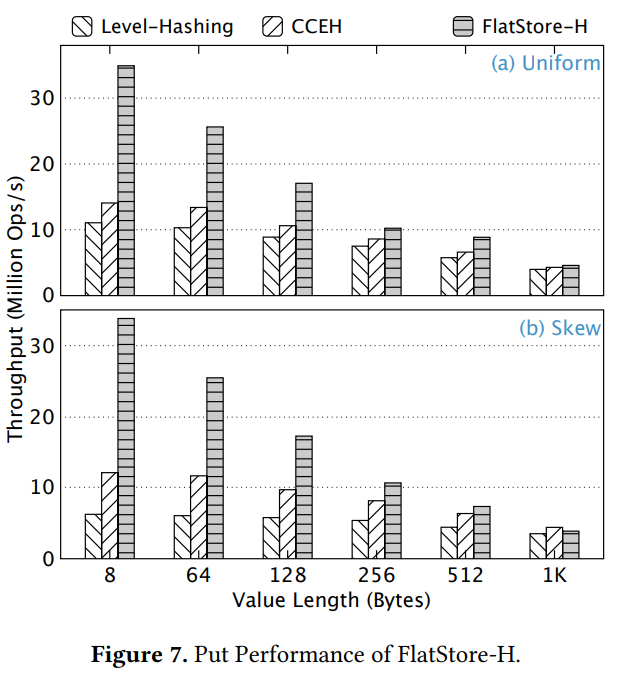
在FlatStore中，由于所有数据都是“异地”更新的，因此在大多数情况下，避免了向同一缓存行的就地刷新。 但是，两个相邻的批次仍可能在OpLog中共享相同的缓存行，因为在构建一个批次时，并非每次我们都准确地收集了16个日志条目（如图底部所示）。 在这种情况下，后一批的持久性会延迟。

为了解决此问题，我们在每个批处理的末尾添加了填充以使它们与缓存行对齐。 对于具有“就地”更新方式的存储系统来说，这是非常困难的，尤其是在运行偏斜的工作负载时。

**惰性持久分配器**。用来存储大型KV项目。由于OpLog已经记录了大型KV的地址，因此不再需要刷新分配元数据（即位图）。 我们通过使用基于多类的分配策略并在运行时延迟存储分配元数据来消除这种冗余。发生崩溃后，可以通过使用OpLog中的指针计算偏移量来恢复位图。

3.1.3 实验结论

选择了四个最新的持久索引方案进行比较。1）基于哈希。Level-Hashing和CCEH 用于与FlatStore-H进行比较，直接使用其默认设置。在实现中，客户端根据密钥哈希将其请求直接路由到特定核心，因此为每个服务器核心创建一个Level-Hashing和 CCEH实例，并删除其中的所有锁定操作。由于哈希表的大小会影响其性能，因此创建的哈希表大小足够大，会在调整大小之前收集性能。与FlatStore-M进行比较。FPTree不是开源的，因此文章基于STX B + -Tree对其进行实现，使用FPTreeFAST＆FAIR所有节点都放置在PM中。与FlatStore-M相似，所有服务器核心共享一个FPTree / FAST-FAIR实例以支持范围搜索。所有比较的索引方案都使用Lazy-persist分配器存储KV记录，而仅在索引中存储一个指针以指向实际数据。

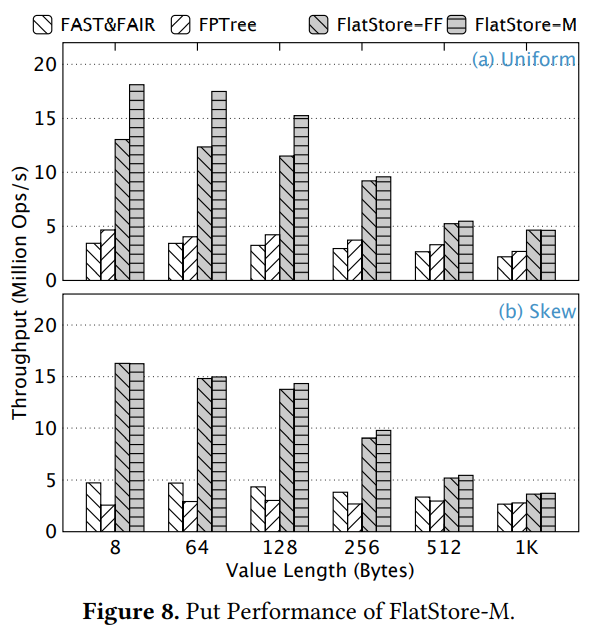


实验结果如图7所示，有以下发现：首先，FlatStore-H的吞吐量比LevelHashing和CCEH高，并且这种性能优势对于小型工作负载尤其明显。对于8字节的值，FlatStore-H可以在统一工作负载下达到35 Mops / s的速度，是CCEH和LevelHashing吞吐量的2.5倍和3.2倍；倾斜工作负载为34 Mops / s，分别比其他两个系统高2.8倍和5.4倍。FlatStore-H的高效性来自以下几个方面：

1）通过引入紧凑的OpLog，FlatStore-H减少了通过简单地添加日志条目，可以为每个Put操作保留的操作数。 取而代之的是，每次更新CCEH和LevelHashing都需要刷新索引条目，存储桶中的位图以及分配器中的实际KV记录。

2）当两个键冲突时，Level-Hashing需要重新哈希相关的条目，而CCEH还需要在段满时分割段，这会放大刷新时间。

3）FlatStore-H有效地将来自多个内核的小型日志条目与流水线HB合并，并以批处理方式保留它们，从而进一步减少了写入次数。 CCEH的性能略高于Level-Hashing，因为它采用了不同的冲突策略，从而减少了冲突的机会。这三个系统对于大型KV都表现出非常接近的性能，因为它们主要受到PM带宽的限制。

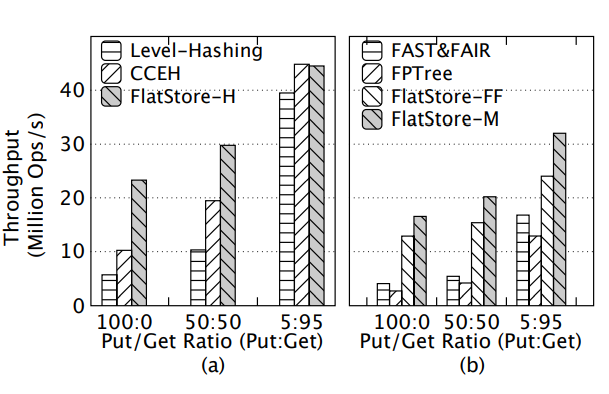


FlatStore-M。为了排除Masstree带来的性能优势，本文还通过用FAST＆FAIR替换Masstree来实现FlatStore-FF。 同样，FAST＆FAIR放在DRAM中以用作易失性索引结构。实验结果如图8所示，本文观察到：

1）FlatStoreM带来的性能改进比FlatStore-H更显着。对于8字节记录，FlatStore-M可以以统一的工作量执行18 Mops/s的速度，是FPTree和FAST＆FAIR吞吐量的3.4倍和4.5倍。 对于偏斜的工作负载，FlatStoreM的吞吐量为16 Mops / s，分别比FPTree和FAST＆FAIR高6.3倍和3.4倍。这主要是因为树节点经常在这些持久索引中合并和拆分，这更可能导致写放大。

2）FlatStoreM具有比FlatStore-FF更高的吞吐量，尤其是对于统一的工作负载。 Masstree旨在提高DRAM中B + -Tree的多核可伸缩性。因此，它比FAST＆FAIR的易失性版本更有效。 即使如此，对于所有评估的工作负载，FlatStore-FF的吞吐量仍然远远高于FPTree和FAST＆FAIR。

3）FPTree在工作负载均匀的情况下显示出比FAST＆FAIR高的吞吐量。这是因为FPTree仅将其叶节点放在PM中，而其他内部节点则直接放在DRAM中。 相反，FAST＆FAIR中的所有树节点都放置在NVM中，从而导致更多的持久性开销。即便如此，FAST＆FAIR在偏斜的工作负载下仍具有更高的性能，因为它结合了更细粒度的锁设计并减少了争用。



本文将Facebook上的ETC池模拟为生产工作量，以评估FlatStore的行为。它具有三峰项目分布，其中项目可以很小（1-13字节），很小（14-300字节）或很大（大于300字节）。

在关键空间（1.92亿）中，40％对应于小项，55％对应于小项，其余5％对应于大项。 本文在细小的项目集上使用zipfian分布（0.99），这代表了许多生产工作负载的强烈偏差。大的项数量少得多，并且表现出高得多的可变性，并且是随机随机选择的。ETC池最初是读取为主的，但文章认为读取为主和写入密集型工作负载分别对应于95：5和50:50的Get：Put比率。作为参考，也给出了100％放置的结果。

3.2 基于混合索引的KV存储引擎

3.2.1 引言

持久键值存储（KV存储）已成为数据中心存储基础结构的重要组成部分。它们被广泛部署在大规模生产环境中，比如搜索引擎，电子商务平台，社交网络等等。在过去十年中，针对KV存储设计和优化进行了大量研究，涉及诸如减少SSD的写入放大等主题，减少索引带来的内存使用，并提高并发性以实现高可伸缩性。常规的KV存储不适用于混合存储系统，因为它们是针对硬盘或SSD的性能特征而设计的。例如，许多现有研究采用日志结构合并树作为索引结构，这避免了对硬盘或SSD的少量随机写入。与硬盘和SSD不同，混合存储系统是字节寻址的，并且为顺序访问和随机访问提供类似的性能。在为混合存储系统设计KV存储器时，保持大粒度的顺序写入会导致NVM的写入放大。

索引的设计是KV存储的基本问题。KV操作（例如Put、Get、Update、Delete和Scan）的效率在很大程度上取决于索引结构的操作效率。例如，搜索B+-Tree索引通常比搜索Hash索引更昂贵，不同索引结构的操作效率差异很大。近来，针对NVM存储器系统提出了许多关于B +树索引的优化。但是，这些技术主要专注于在NVM中直接保留B+-Tree索引时达到降低一致性成本的目的。另一方面，KV存储的可伸缩性受到索引结构的可伸缩性的限制。例如，对Hash索引进行分区允许将索引结构扩展到多个线程，但是对B+-Tree索引进行分区会在拆分大分区或合并小分区时导致昂贵的数据移动。因此，我们认为在混合存储器上设计KV存储的索引结构仍然是有可操作性的。

本文提出了HiKV，它是在混合内存上运行的基于混合索引实现的KV存储。HiKV背后的中心思想是采用混合索引：将哈希索引放置并保留在NVM中，而将B+-Tree索引放置在易失但快速的DRAM中。混合索引充分利用了两个索引的独特优点。它保留了散列操作的固有效率，以支持单键操作（Get、Put、Update和Delete）。而且，它有效使用B+-Tree中的排序索引来加快扫描速度。

采用混合索引带来了许多挑战。首先，在为某些KV操作提供服务时，由于HiKV需要更新两个索引以使其保持一致，因此增加了延迟。HiKV通过在快速地DRAM中放置慢速B+树索引，在慢速NVM中放置快速Hash索引来解决此问题。此外，HiKV异步更新B+-Tree索引以进一步隐藏延迟。其次，混合索引的可伸缩性需要进一步设计。对散列索引进行分区可提供良好的可伸缩性，而对B+-Tree索引进行分区会因数据迁移而带来高昂成本。因此，HiKV采用分区Hash索引和全局B+-Tree索引。HiKV将硬件事务存储应用于B+-Tree索引的并发控制，并且细粒度的锁定可支持单个Hash索引分区中的并发访问。最后，保证混合索引的崩溃一致性会导致对NVM的昂贵写入。HiKV采用选择性一致性，只能通过有序写入来确保Hash索引和KV项的一致性。HiKV将B+-Tree索引保留在DRAM中，并在系统故障后重建它。

文章实现的HiKV和基于NVM的最新KV存储NVStore、FPTree进行了比较实验。我们使用微基准和广泛使用的YCSB评估了三个KV存储系统。对于微基准，与NVStore和FPTree相比，HiKV可以分别将延迟降低54.5％至83.2％和28.3％至86.6％。 对于YCSB工作负载，HiKV的吞吐量要比NVStore高出1.7倍至5.3倍，而FPTree则高出24.2％至6.4倍。

3.2.2 主要设计

基本的KV操作包括Put、Get、Update、Delete和Scan。要找到请求的KV项，单键操作首先要精确地使用一个键来搜索索引。找到KV项后，Get将直接返回数据，而写操作（Put、Update或Delete）要求保留更新的索引条目和新的KV项。因此，索引搜索和数据持久化的效率对于这些操作很重要，Hash索引固有地支持高效搜索。此外，考虑到NVM读取的性能与DRAM中的类似，在NVM中将哈希索引作为混合索引的一部分是一种合理的设计选择。这种设计不仅保留了对哈希索引的快速搜索，而且还允许直接在NVM中保留索引，而无需将额外的数据从DRAM复制到NVM。

另一方面，Scan操作将开始键和计数（或开始键和结束键）作为输入，这会从排序索引中受益。为了有效地支持Scan，混合索引在主内存系统中采用了广泛使用的B + -Tree索引。为了保持一致的混合索引，根本上需要更新KV写入的哈希索引和B + -Tree索引。 由于叶节点的排序以及拆分、合并，因此更新B + -Tree索引涉及许多写入操作。因此，文章将B+-Tree索引放置在快速DRAM中，以避免在混合内存中写入到慢速的NVM中。

在处理KV写入时，HiKV需要同时更新哈希索引和B+-Tree索引，以使其保持一致状态。一种直观的解决方案是同步更新两个索引。由于树特定结构的操作昂贵，例如搜索、排序、拆分和合并，因此B + -Tree索引的同步更新会增加KV写入的额外延迟。因此，HiKV将异步更新用于混合索引。换句话说，HiKV会保留对KV项目和NVM中的哈希索引的同步更新。对于DRAM中的B + -Tree索引，HiKV在后台异步更新它以隐藏额外的延迟。

服务线程负责将KV项目写入NVM，然后将新添加的索引条目写入哈希索引。 最后，服务线程将Put请求插入到更新队列中，然后返回。异步线程（后端线程）从更新队列获取请求，并在后台操作B+-Tree索引。如果由于系统崩溃而无法更新B+-Tree索引，HiKV可以从哈希索引中恢复B+-Tree索引。这样，可以减少观察到的KV写入的等待时间。

3.2.3 实验结论

将HiKV与最新的NVM KV存储NVStore和混合内存KV存储FPTree进行比较。没有将HiKV与基于磁盘的KV存储（例如RocksDB）进行比较。这是因为HiKV是为字节可寻址NVM设计的，其I/O堆栈与RocksDB的I/O堆栈完全不同。我们将根据其论文中的描述尽可能地重新实现NVStore和FPTree。

我们首先使用微基准评估HiKV的单线程性能。对于仅读取数据（包括Get和Scan）的基准，所有四个KV存储都使用一个线程。HiKV被设计为在提供写请求时采用服务线程和后端线程一起操作B + -Tree索引。因此，对于放置、更新和删除，HiKV配置为使用一个服务线程和一个后端线程。为了公平比较，NVStore，FPTree和FPTree C都配置有两个线程。

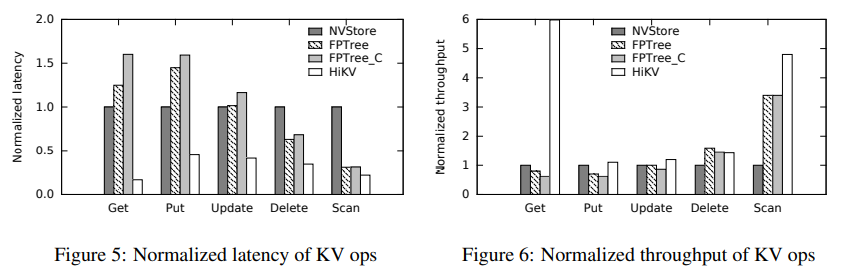


图5显示了HiKV的等待时间减少。对于Get，HiKV可以分别比NVStore和FPTree

减少83.2％和86.6％的延迟。HiKV只需要查找快速哈希索引。但是，NVStore和FPTree不仅需要查找树索引，而且还需要顺序查找叶节点，因为叶节点中的键未排序。对于Put、Update、Delete，HiKV可以将延迟比NVStore减少54.5％、58.4％和65.3％，比FPTree减少68.8％、59.1％和45.0％。

图6显示了HiKV的吞吐量提高。HiKV可以将吞吐量比NVStore提高5.0倍和3.8倍，比FPTree的Get和Scan提高6.4倍和41.2%。对于Put和Update，HiKV分别优于NVStore和FPTree 10.4％/ 19.6％和55.9％/ 19.6％。HiKV的Delete吞吐量比NVStore的Delete吞吐量高43.2％，比FPTree的Delete吞吐量低10.0％。

3.3 写优化和高性能哈希索引方案

3.3.1 引言

随着DRAM技术在密度缩放和功率泄漏方面面临巨大挑战，非易失性存储器（NVM）技术，例如ReRAM，PCM，STT-RAM和3D XPoint，是构建未来内存系统的候选。非易失性使数据可以持久存储到NVM中，以进行瞬时故障恢复。由于字节寻址和接近DRAM的访问延迟，因此可以使用CPU加载和存储指令通过内存总线直接访问持久性存储器，从而避免了传统的基于块的接口的高开销。但是，NVM通常会受到有限的耐用性和低写入性能的困扰。

存储器体系结构和特性的重大变化导致以传统方式索引数据的效率低下，从而忽略了数据一致性和新的NVM设备属性的要求。大量现有工作已经改进了基于树的索引结构，以有效地适应持久性存储器，例如CDDS B树、NV树、wB +树、FPTree、WORT和FAST＆FAIR。基于树的索引结构通常具有平均O（log（N））的查找时间复杂度，其中N是数据结构的大小。与基于树的索引结构不同，基于散列的索引结构是平面数据结构，可以实现恒定的查找时间复杂度，即O（1），独立于N。由于提供了快速的查找响应，因此哈希索引结构在主存储系统中得到了广泛的应用。例如，它们是主内存数据库中的基本组件，并用于索引内存中的键值存储，例如Redis和Memcached。但是，当哈希索引结构保存在持久性内存中时，存在许多非平凡的挑战，这些挑战很少被现有工作所触及。

高开销，以确保一致性。发生系统故障时，永久性存储器中的数据结构应避免任何数据不一致（即数据丢失或部分更新）。但是，通过内存总线直接访问NVM的新体系结构会导致高开销以保证一致性。首先，内存写入通常由CPU和内存控制器重新排序。为了确保为了保证一致性，必须采用高速缓存行刷新和内存防护，从而引入高性能开销。其次，现代处理器的原子写入单元通常不大于内存总线宽度（例如，对于64位处理器为8字节）。 如果写入的数据大于原子写入单元，则需要采用昂贵的日志记录或写时复制（CoW）机制来保证一致性。降低写入性能。与读取相比，NVM中的内存写入消耗有限的耐久性，并导致更高的等待时间和能量。此外，对持久性内存的更多写入还会导致更多的缓存行刷新和内存隔离以及可能的日志记录或CoW操作，从而大大降低系统性能。 因此，在NVM中减少写操作很重要。先前的工作证明了常见的哈希方案，例如链式哈希，跳房子哈希和布谷鸟哈希通常会导致许多额外的内存写入，以处理哈希冲突。提出了诸如PFHT和路径散列之类的写友好散列方案，以减少散列索引结构中的NVM写操作，但其代价是降低了访问性能（即搜索，插入和删除操作的吞吐量）。调整哈希表大小的成本效率低下。随着哈希表的负载因子（即存储项目数与总存储单元数之比）的增加，哈希冲突数增加，导致访问性能下降以及插入失败。调整大小对于哈希表在其负载因子达到阈值或发生插入失败时增加大小至关重要。

调整哈希表的大小需要创建一个新的哈希表，其大小通常是原来的两倍，然后迭代地将旧哈希表中的所有项目重新哈希为新哈希表。

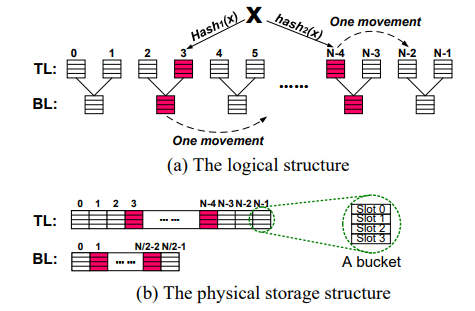
调整大小是一项昂贵的操作，因为需要O（N）时间复杂度才能完成，其中N是哈希表中的项目数。调整大小还需要进行N次插入操作，从而导致大量NVM写操作发生，并在持久性内存中进行了缓存行刷新和内存隔离。

为了解决这些挑战，本文提出了级别哈希，一种写优化的高性能哈希索引方案，该方案具有低开销一致性保证和经济高效地调整持久内存大小。

3.3.2 主要设计

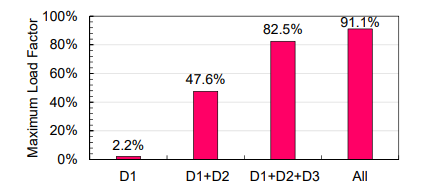
文章提出了级别Hash，一种写优化的高性能散列索引方案，该方案具有经济高效的大小调整和对持久性内存的低开销一致性保证。写优化的哈希表结构的级别哈希表是一种新的开放式寻址结构，具有BCH，PFHT和路径哈希的所有优点，包括内存高效，写优化和高性能，同时通过执行避免了它们的缺点，以下为主要设计决策。

每个存储桶多个插槽。根据Facebook和百度发布的多个键值工作负载特征，大小小于缓存行大小的小键值项在当前键值存储中占主导地位。例如，大多数键的大小都小于32个字节，而Facebook的键值存储中常见的请求类型是带有2字节值的16或21字节键。受现实工作负载特性的影响，我们通过在每个存储桶中设置多个插槽（例如每个存储桶4个插槽）来使级别哈希表具有缓存效率，如图所示。



因此，存储桶可以在一个插槽中存储多个键值项。当访问级别哈希表中的存储桶时，可以在一个内存访问中将同一存储桶中的多个键值项预取到CPU缓存中，从而提高了缓存效率，从而减少了内存访问次数。

每个密钥有两个哈希位置。由于每个存储桶都有k个时隙，因此哈希表最多可以处理单个哈希位置中发生的k-1个哈希冲突。可能有超过k个键值项被散列到同一位置。 在这种情况下，容易发生插入失败，从而导致低负载率。 为了解决这个问题，我们通过使用两个不同的哈希函数（即hash1和hash2）使每个键具有两个哈希位置，例如BCH，PCHT和路径哈希。一个新的键值项被插入到两个哈希位置之间较少负载的存储桶中。由于两个独立哈希函数的随机化，哈希表的负载因子得到了显着改善，如图所示。



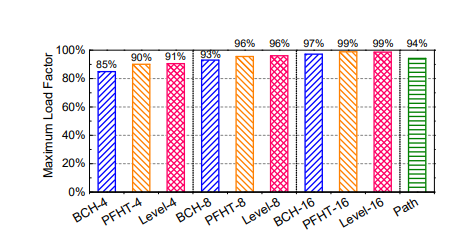
基于共享的二级结构。级别哈希表中的存储桶分为两个级别，即顶层和底层。哈希函数只能寻址最顶层的存储桶。底层不可寻址，用于为顶层提供备用位置以存储冲突的键值项目。每个底层存储桶由两个顶层存储桶共享，因此底层存储空间的大小是顶层存储空间的一半。如果在顶级存储桶中发生哈希冲突并且该存储桶中的所有位置都已被占用，则有冲突的键值项可以存储在其相应的底层存储桶中。通过使用两级结构，哈希表的负载因子得到了显着改善，如图所示。此外，由于每个可寻址桶都具有一个备用桶，因此搜索操作最多只需要探测四个桶即可，在最坏的情况下扩展时间复杂度。

每次成功插入最多一个动作。为了使键值项能够在存储桶之间平均分配，如果在BCH中插入两个存储桶时两个存储桶都已满，则BCH会逐出现有项之一，从而导致级联写入，即对NVM不友好。为了避免级联写入的问题，级别散列允许每次插入最多移动一项。具体来说，在插入新项目期间，如果两个顶级存储桶已满，我们将检查是否有可能将任何键值项目从其两个顶级存储桶之一移至其替代的顶级位置。如果无法移动，文章进一步将新项目Inew插入底部。如果两个底层存储桶已满，还将检查是否有可能将任何键值项从其两个底层存储桶之一移至其备用底层存储位置。如果移动仍然失败，则插入失败，并且需要调整哈希表的大小。

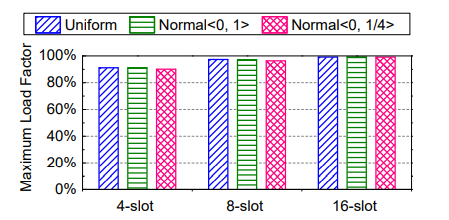
3.3.3 实验结论

最大负载因子是哈希表的重要指标，因为它直接影响哈希表可以存储的键值项的数量和硬件成本。为了评估最大负载因子，我们将唯一的字符串键插入空的BCH，PFHT，级别和路径哈希表中，直到发生插入失败。具体而言，当单次插入操作在500次驱逐后未能找到空插槽时，BCH达到最大负载因子。对于PFHT，将哈希表总大小的3％空间用作存储，遵循原始论文中的配置。

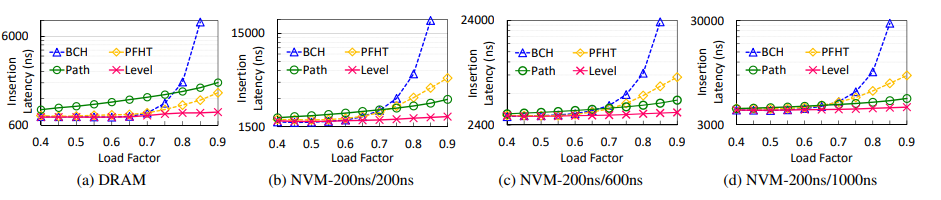
当存储柜已满时，PFHT达到最大负载系数。当单次插入未能找到空的插槽或存储桶时，级别和路径哈希表将达到最大负载因子。



图显示所有四个哈希表都可以达到最大负载率的90％以上。图还比较了不同的哈希表和每个存储桶中不同数量的插槽。每个存储桶中的更多插槽会为BCH，PFHT和级别哈希表带来更高的最大负载因子。对于每个存储桶中相同数量的时隙，PFHT和级别哈希表具有大约相同的最大负载因子，高于BCH。路径哈希表是一个单项存储桶表，可实现高达94.2％的最大负载率。



我们还评估了具有不同分布整数键（包括均匀和偏斜的正态密钥分布）的级别哈希表的最大负载因子，如图所示。我们观察到，级别哈希表针对不同的密钥分布实现了近似的最大负载因子。原因是哈希函数将键映射到随机哈希值，因此无论键分布是什么，生成的哈希值分布仍然是随机的。然后，根据哈希值在哈希表的各个存储桶中随机分配密钥。因此，歪斜的密钥分布不会导致歪斜的哈希值。



图显示了DRAM中不同哈希表中每个插入操作的平均等待时间和持久存储器中不同哈希表的平均插入等待时间。与a和b中的实验结果相比，我们观察到持久性存储器中的插入等待时间远高于DRAM中的插入等待时间，而持久性存储器的读/写等待时间（200ns）与DRAM（136ns）接近。主要原因是必须通过CLFLUSH将每个插入的项目刷新到持久性内存中，并通过MFENCE确保写入顺序以确保一致性，从而显着增加了延迟。

4 结论

现有的KV存储在处理当今流行的工作负载时会生成大量的小型写入，这与PM中的持久性粒度不匹配。 为了解决此问题，文章[1]将KV存储分离为易失性索引和日志结构存储，从而提出了一个名为FlatStore的基于PM的KV存储引擎。它使用每核OpLog管理索引和小型KV，以便更好地支持批处理。还建议使用流水线水平批处理来提供高吞吐量和低延迟的请求处理。评估表明，FlatStore明显优于现有的KV存储。持久键值存储已广泛部署在实际应用程序中。由DRAM和NVM组成的混合内存允许存储系统将数据直接保存在内存中。 建立面向混合内存的KV存储可以利用其性能特征。当今的应用程序非常需要有效地支持丰富的KV操作（输入/获取/更新/删除/扫描）。但是，现有KV存储使用的哈希索引或B+-Tree索引不能有效地支持所有这些操作。在文章[2]中，提出了一种混合索引，它在NVM中采用哈希索引来进行快速搜索和直接持久化，而在DRAM中采用B+-Tree索引来进行快速更新和支持范围扫描。在混合索引的基础上，文章构建了HiKV，这是一个基于混合索引的键值存储，具有良好的可伸缩性和崩溃一致性保证。为了有效地索引持久性存储器上的数据，文章[3]提出了一种写优化的高性能哈希索引方案，称为级别哈希，以及一种经济高效的就地调整大小方案和无日志一致性保证计划。级别哈希仅通过使用细粒度锁定即可有效地支持多读取器和多写入器并发。

5 参考文献

[1] Chen Y , Lu Y , Yang F , et al. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. ASPLOS '20: Architectural Support for Programming Languages and Operating Systems. 2020.

[2] F Xia,D Jiang,J Xiong,N Sun,et al. HiKV A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems.USENIX ATC '17: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference.July 2017.

[3] P Zuo,Y Hua,J Wu.Write-optimized and high-performance hashing index scheme for persistent memory. OSDI'18: Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation.October 2018