O^3-LSM：通过三层卸载最大化解耦式LSM写入性能

Qi Lin
亚利桑那州立大学
USA
qlin36@asu.edu

Gangqi Huang
亚利桑那州立大学
USA
alexhuang1403@gmail.com

Te Guo
普渡大学
USA
guo777@purdue.edu

Chang Guo
亚利桑那州立大学
USA
cguo51@asu.edu

Viraj Thakkar
亚利桑那州立大学
USA
viraj.dt@asu.edu

Zichen Zhu
Google
USA
zczhu@bu.edu

Jianguo Wang
普渡大学
USA
csjgwang@purdue.edu

Zhichao Cao
亚利桑那州立大学
USA
zhichao.cao@asu.edu

arXiv:2603.05439v1 [cs.DB] 5 Mar 2026

摘要

基于日志结构合并树的键值存储（LSM键值存储）已通过压实卸载等技术进行优化和重新设计，以减少计算资源与存储资源之间的网络 I/O。然而，计算节点内存空间受限且刷新速度缓慢，严重限制了现有优化方案的整体写入吞吐量。本文提出了**O^3-LSM**，一种全新的基础性 LSM键值存储架构。该架构利用共享的解耦内存（DM）支持三层卸载，即内存表**卸**载、刷新**卸**载，以及现有的压实**卸**载。与仅采用压实卸载的现有解耦 LSM键值存储相比，O^3-LSM通过解决上述问题，最大限度地提升了写入性能。

O^3-LSM首先利用一种新颖的解耦内存优化型内存表实现动态内存表卸载，既扩展了写缓冲区，又支持快速、异步且并行的内存表传输。其次，我们提出协同刷新卸载，将刷盘控制平面与执行解耦，并通过专用调度和全局优化，支持在任意节点进行内存表刷新卸载。第三，O^3-LSM进一步采用分片级优化加以改进：依据互不相交的键范围将内存表划分为多个分片，各分片可独立传输和刷新，从而实现跨分片并行。此外，为缓解解耦环境中的查找缓慢问题，O^3-LSM还采用自适应的缓存增强型读取委派机制，将精简的本地缓存与解耦内存辅助的内存表委派读取相结合。评估结果表明，与解耦式RocksDB、CaaS-LSM和Nova-LSM相比，O^3-LSM的写入、范围查询和点查找吞吐量最高分别提升**4.5**X、**5.2**X和**1.8**X，P99延迟最高降低**76%**。

1 引言

随着云计算和高速网络的快速发展，解耦数据中心（**解耦数据中心**）已在业界得到广泛采用，以应对传统单体架构中资源利用率、负载均衡和可扩展性方面的挑战。单体架构 [31, 44, 52, 53]。解耦数据中心通过将计算资源、内存和存储资源解耦为可独立扩展的资源池（例如计算集群、解耦存储（**存储设备**）和解耦内存（**解耦内存**）），实现了高度灵活性。在这一架构演进的推动下，**基于日志结构合并树的键值存储（LSM键值存储）**正在针对解耦数据中心进行重新设计，从而形成解耦式 LSM键值存储，以充分利用解耦数据中心模型，最大限度地提升资源利用率和性能。

在典型的解耦LSM键值存储中，写入的键值对（KV-pairs）最初会被批量缓存在一个或多个计算节点（**计算节点**）的写缓冲区中，形成内存表。当内存表变满时，会被标记为不可变，并通过网络从计算节点刷新到存储设备，成为一个排序字符串表（SST）文件。为了提高空间效率和读取性能，存储设备中的 SST 文件会定期通过网络读回计算节点，合并为新的 SST 文件，再写回存储设备。这种定期合并也称为压实，在后台进程中运行，其设计目标是不干扰前台用户写入进程。解耦数据中心所提供的解耦特性天然适合这项任务，因为它允许**将压实进程与计算节点分离，并卸载到存储设备**，从而显著降低计算节点上的资源竞争和网络流量。因此，许多最先进的解耦LSM键值存储——包括 Meta 的 Disaggregated-RocksDB [17], 、字节跳动的 TerarkDB [7], 、IS-HBase [9], 、Hailstorm [6],、Nova-LSM [25], 、Rocks-Cloud [39], 和 CaaS-LSM [51] ——都采用了这一设计，将压实卸载到存储节点，或将其重新分配到各计算节点，以减轻压实期间计算节点与存储设备之间的数据移动所导致的写入性能损失。

然而，压实卸载并不能有效缓解写入减速和停顿，而这些问题是由**写缓冲区内存有限以及刷新操作缓慢**导致的。在数据写入过程中，当达到写缓冲区上限时，通常会对新写入进行限流或阻塞，直到通过刷新释放内存空间，这会直接导致写入变慢和停顿。鉴于 LSM-KVS实例通常需要在读取路径（即块缓存）和写入路径（即内存表）上占用大量内存 [10], 计算节点的内存

通常是有限的，尤其是在解耦数据中心设置中，单个计算节点可能同时承载数十个 LSM-KVS 实例及其他应用。除了内存有限之外，内存表刷新还会对具有高延迟的存储设备产生大量网络 I/O [9, 17, 51]。这些网络流量会与计算节点上的其他后台和前台 I/O（例如预写日志追加/同步、压实和 SST 文件读取）争用，从而延长刷新过程，并进一步导致写入减速和停顿 [50]。此外，如果由于串行且缓慢的 L_0 压实 [5, 50]导致第 0 层的 SST 文件数量（L_0）达到预定义上限，刷新操作也可能被暂停 。

借助 RDMA 等广泛部署的远程内存技术， [36], 解耦数据中心能够为计算节点提供共享的解耦内存（DM）池，作为内存的第二层。鉴于这一能力，解决写缓冲区受限问题的直接方法，是将内存表卸载到 DM，作为短期暂存区，从而扩展解耦 LSM键值存储的写缓冲区，而不是立即刷新到存储设备。然而，**仅将内存表卸载到 DM，反而会降低写入吞吐量并增加读取延迟**。

问题 1：远程内存表传输与重建代价高昂。即使 RDMA 速度很快，访问远程内存仍然比访问本地 DRAM 慢，因此将内存表移至 DM 并非简单的内存复制。内存表需要经过远程网络接口卡、PCIe 和互连网络，从而带来不可忽视的传输延迟和带宽消耗。此外，内存表包含大量指针（例如，从内存区域分配的跳表节点），因此直接复制原始字节会使 DM 中的地址失效。要使其在 DM 上可供查询，还需额外更新指针并重建索引。因此，在 DM 上进行内存表传输和重建都会带来明显的性能损耗。**问题 2：内存表刷新效率低下**。将不可变内存表卸载到 DM 只是第一步。对于解耦 LSM键值存储，内存表仍需刷新到存储设备以实现持久化。然而，DM 无法直接将这些卸载的内存表刷新到存储设备，因为刷新需要进行键值对排序与合并、压缩、SST 文件构建和 Manifest 更新，而 DM 不具备这些功能。因此，必须将内存表读回计算节点，再由所属的 LSM 键值存储在本地执行刷新。这个过程增加了一次网络跳数，重新引入序列化和反序列化开销，而且还可能受到同一计算节点上其他前台和后台操作所造成的 I/O 争用的进一步影响。与此同时，随着大量驻留于 DM 的内存表被刷新到存储设备， L_0 SST 文件不断累积，加剧了 L_0压实缓慢的问题。所有这些因素都会降低整体性能。

问题 3：在 DM 上搜索内存表的速度较慢。将内存表卸载到 DM 后，搜索它们可能会导致读取性能显著下降。一方面，由于内存表采用半排序且指针密集的结构（例如跳表或树），在单个内存表中查找一个键需要多次内存访问。当内存表位于 DM 时，这一过程会涉及多次远程内存访问往返，从而导致更高的查找延迟。另一方面，一次读取查询可能需要搜索 DM 上属于该 LSM键值存储的所有累积内存表。随着 DM 上累积的内存表越来越多，读取性能可能会变得更差。

为应对将内存表卸载到解耦内存所面临的上述挑战，我们提出 **O^3-LSM**，这是一种全新的基础性 LSM键值存储架构，利用共享的解耦内存（基于远程直接内存访问）支持三层卸载，涵盖内存表**O**卸载、刷新**O**卸载以及现有的压实**O**卸载。与仅支持压实卸载的最先进解耦LSM键值存储相比，O^3-LSM 通过引入以下四项关键创新实现了更优的性能：

• **DM优化内存表**：O^3-LSM 引入了一种针对 DM 优化的内存表，旨在无需大量重建即可直接在 DM 上搜索。O^3-LSM 不会在远程 DM 上传输并重建指针密集型内存表，而是以适合 DM、内存连续的布局写入内存表，并采用索引与数据分离，使卸载成为低成本的数据传输，且内存表可以直接在 DM 上查询。

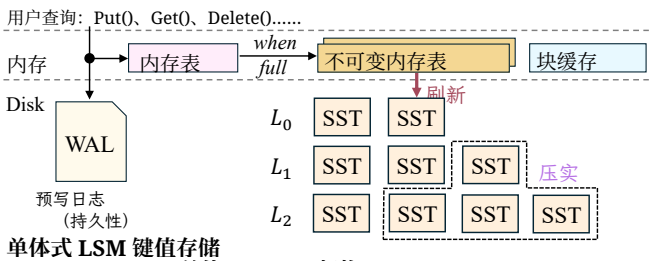
• **协同刷新卸载**：O^3-LSM 设计了一种轻量且经过审慎设计的多阶段卸载协议，可在具有解耦内存访问能力的任意节点上执行内存表刷新。该协议支持多种执行模式（**本地、解耦内存内、远程计算节点**）以及各类控制消息，为**感知争用的刷新调度器**奠定了基础。该调度器通过根据工作负载、数据局部性和资源利用率动态调整，并处理潜在故障，协调所有刷新操作，从而在确保原子性和隔离性的同时，最大限度地提升整体写入性能。

• **异步分片**：O^3-LSM 还将每个内存表划分为一组互不重叠的键范围分片，具体做法是将键值块拆分为多个块，从而支持向解耦内存进行异步、高度并行的传输。在刷新多个内存表时，O^3-LSM 会聚合不同内存表中键范围相同的键值分片块。这种方法生成互不重叠且与分片对齐的 L_0 SST 文件，有效地将串行的 L_0压实拆分为多个并行任务，从而显著减少整体写入停顿。

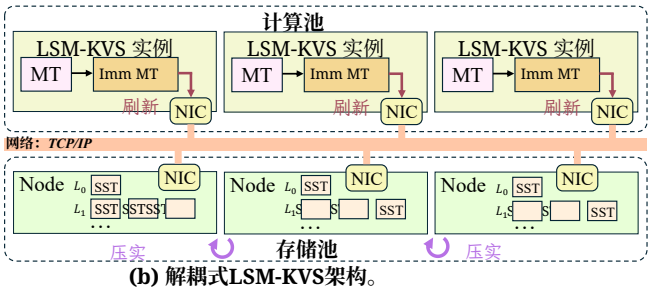
• **缓存增强型读取委派**：O^3-LSM通过自适应读取委派加速读取。借助DM优化内存表结构，O^3-LSM在计算节点维护一个小型键偏移缓存，从而按每次读取决定读取路径。该自适应机制决定是使用单边远程直接内存访问_READ直接从解耦内存检索数据，还是通过双边远程直接内存访问_SEND将读取委派给解耦内存，从而减少不必要的远程网络传输，并缓解计算节点额外的内存占用。

实现与总体评估结果。 我们基于 RocksDB v8.2.0 对 O^3-LSM 进行了原型实现，修改了 30,000 行代码，并已在**GitHub**[1]上开源。我们在 CloudLab [19] 上，在相同写缓冲区限制下，将其与最先进的 LSM键值存储方案进行对比评估，包括 Disagg-RocksDB [17],、CaaS-LSM [51], 和 Nova-LSM [25] 。在多种工作负载下，O^3-LSM 在吞吐量和 P99延迟方面均取得性能提升。对于随机写工作负载，O^3-LSM 的吞吐量分别提升**4.5**X、**3.4**X和**4.4**X，同时 P99延迟最高降低**60%**，优于 Disagg-RocksDB、CaaS-LSM 和 Nova-LSM。对于随机读工作负载，与这些基线系统相比，O^3-LSM 的吞吐量分别提高**1.8**X、**0.6**X和**0.3**X，P99延迟最高降低**69%** 。对于范围查询工作负载，O^3-LSM 的吞吐量最高提升**5.2**X，同时 P99延迟最高降低**22%**。即使在混合工作负载（50% 读操作和 50% 写操作）下，O^3-LSM 的吞吐量也分别最高提升 **3**X、**2.3**X和**1.9**X，同时 P99延迟最高降低**76%**。对于真实世界的

[1]https://github.com/argetterxx/O3-LSM



(a) 单体LSM-KVS架构。



(b) 解耦式LSM-KVS架构。

图 1：单体式与解耦式 LSM-KVS 架构的比较。

应用 Kvrocks [4], 中，O³-LSM 的吞吐量最高提升**3.4**X，同时P99延迟最高降低**54%**。

2 背景

2.1 基于 LSM 的键值存储

RocksDB [16] 和 LevelDB [22] 等 LSM 键值存储通过日志结构化存储来优化高性能写入。如图 1a 所示，LSM 键值存储会将键值对追加到预写日志（WAL）和内存活动内存表中。已满的活动内存表会变为不可变，并通过刷新过程序列化为排序字符串表（SST）文件，写入 L_0。因此，每个 L_0 SST 文件都可能包含跨越整个键空间的键值对。SST 文件分布在多个层级中。L_0 文件会依次压实到 L_1，因为它们的键范围通常会重叠。相比之下，层级L_n （$n \geqslant 1$）的键范围互不重叠，因此可以与 L_{n+1}中存在重叠的文件并行压实。为管理背压，如果不可变内存表或 L_0 文件过多，LSM 键值存储会触发写入减速或停顿。对于读取查询，LSM 键值存储会在内存中维护块缓存，以缓存 SST 文件中的数据块和元数据块。

2.2 基于解耦式LSM的键值存储

解耦数据中心（DDC）通过将计算资源、内存和存储资源解耦为由高速互连网络连接的独立资源池，提升资源利用率和弹性，因此日益受到关注 [3, 17, 53]。针对解耦数据中心优化 LSM 键值存储（称为解耦LSM键值存储），尤其是将持久化数据（例如 SST 文件）存储在解耦存储设备上（例如 Meta 的 Tectonic [37], 、Google 的 GFS [23], 和 HDFS [42]），已成为一种流行做法 [17, 51]。如图 1b 所示，典型的解耦LSM键值存储设计会将内存表保留在计算节点（CN）上，并将 SSTable 存储在存储节点（DSN）上，计算节点通过文件或块接口访问存储节点。这种分离有助于实现更好的负载均衡，同时保持与本地部署相当的写入性能 [17]。近期研究对运行在存储设备上的解耦LSM键值存储进行了优化，主要聚焦于压实卸载 [6, 7, 17, 51]。

然而，计算节点内存方面仍存在瓶颈。LSM键值存储在读取路径（块缓存）和写入路径（内存表缓冲区）上都需要大量内存 [10]. 计算节点通常承载数十个 LSM-KVS 实例，因此难以为每个实例分配大量内存 [20]。LSM-KVS 实例内存不足会触发更频繁的刷新操作，以便快速持久化内存表。这些刷新会放大发往存储设备的网络 I/O，而更高的延迟和共享带宽会延长该过程，并可能导致写入减速或停顿。该问题会因 L_0 压实而加剧，

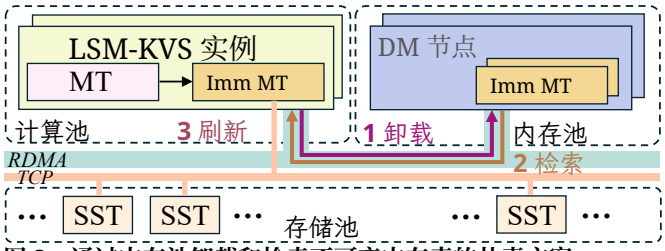


图 2：通过内存池卸载和检索不可变内存表的朴素方案。

指标	K=2	K=4	K=8	K=16	K=32
由内存表引发的写入停顿（次数）	737	614	268	115	34
停顿总计占比（%）	96	85	60	30	11
写停顿延迟 P99（毫秒）	2	6	10	14	18

表 1：解耦内存卸载内存表对写入停顿的影响

其中 L_0 文件覆盖整个键空间。因此，L_0 文件只能按顺序压实，而较高的写入压力很容易命中 L_0 文件上限，从而导致限流或阻塞刷新 [5, 50]。因此，内存压力和缓慢刷新仍是当前解耦式 LSM-KVS 设计中的核心瓶颈。

2.3 解耦内存的机遇：内存表的新层级

远程直接内存访问 [34] 和 CXL [11] 已在解耦数据中心中实现了解耦内存（DM），提供了内存池化和共享。与存储设备不同，DM 通常不具备持久化特性，以最大限度地提升性能。研究人员已经探索了在 DM 上运行的内存型 LSM-KVS，例如 dLSM [46], ，它将 SSTable 存储在 DM 上，并在计算节点中缓存内存表。尽管 dLSM 缺乏持久化保证，且不同于 Disaggregated-RocksDB 或 CaaS-LSM，但 DM 仍然是缓解写入路径内存压力、提升基于持久化存储设备的系统性能的有前景底层基础设施。

除了写入路径上的机会外，DM 在读取路径行为上也不同于本地内存。基于 RDMA 的 DM 通常采用两种方法。单边 RDMA_READ 允许计算节点利用远程地址和远程密钥直接拉取字节，从而绕过远程 CPU，但可能产生额外的往返次数。另一种方法是双边 RDMA_SEND/RDMA_RECV：远程线程读取本地 DRAM，并通过一次交互完成响应，从而减少往返次数，但会消耗远程 CPU 并增加 RPC 开销。实践中，单边读取优先考虑零 CPU 占用，而非网络延迟；双边读取则以消耗远程 CPU 为代价，尽可能减少往返次数。

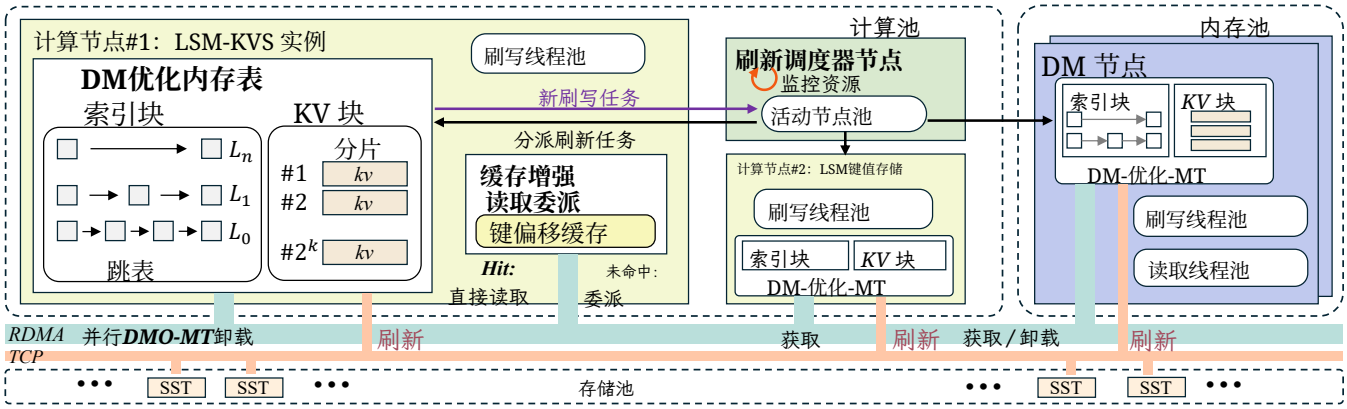


图 3：O^3-LSM 的整体架构。

3 动机与挑战

解耦存储（DS）为持久化 LSM 键值存储提供弹性存储容量，但计算节点内存仍然是瓶颈，尤其是在共置实例较多时。解耦内存（DM）可以作为计算节点内存与 DS 之间的低延迟、高带宽层。因此，利用 DM 层缓存额外的内存表，同时将 SST 文件的持久性保留在 DS 上，是一种有望缓解每个实例内存压力、实现高性能且不牺牲持久化的方案。

3.1 朴素方案

缓解计算节点（计算节点）的本地内存压力，一种直接的方法是将不可变内存表卸载到解耦内存，使解耦内存成为计算节点内存的快速、可扩展扩展（图 2）。我们通过修改解耦 RocksDB [17]，使其通过远程直接内存访问传输内存表，同时将预写日志保留在本地、将 SST 保存在 HDFS [42]，对此进行了评估。我们使用 db_bench 的“随机写入”测试，处理 116B 键值对，并将远程内存表上限（K）从 2 提高到 32。如表 1 所示，这使内存表导致的写入停顿减少了 95%。

然而，吞吐量从 119 kop/s 降至 87 kop/s，p99 写停顿延迟则从 2 ms 上升至 18 ms。分析表明，尽管停顿减少了，但在解耦内存上迁移和重建内存表耗费了总时间的 18.7%。此外，从解耦内存检索内存表以便刷新到存储设备，相较于本地刷新增加了 7.8% 的开销。读取性能也有所下降：从解耦内存读取需要 29 us，而本地读取只需 4.7 us。该延迟主要由远程直接内存访问开销造成，其中远程直接内存访问_SEND 耗时 12 us，远程直接内存访问_RECV 耗时 11 us。远程搜索还会产生额外的复制开销。这些结果表明，迁移、重建和检索开销可能超过减少写入停顿所带来的收益。

3.2 问题与挑战

如第 3.1 节所述，将内存表从计算节点本地内存卸载到解耦内存，可能会进一步提升解耦LSM键值存储的性能。然而，要解决以下性能问题颇具挑战：1）内存表传输和重建带来的延迟会导致写入变慢；2）从解耦内存检索内存表会延迟刷新；3）在解耦内存中搜索内存表会导致读取性能下降。

如何高效地将内存表从计算节点传输并重建到解耦内存？ 由于远程内存访问延迟可能是本地 DRAM 访问延迟的数倍，[2, 21, 41]，将内存表从计算节点传输到解耦内存可能导致较高的延迟，并阻塞后续的内存表写入。在写密集型工作负载下，情况可能会更糟。此外，由于内存表的数据结构设计（例如广泛使用的基于跳表的内存表），内存表包含大量指针。将内存表传输到解耦内存后，所有指针都会失效，需要在解耦内存中重新分配，这可能会很耗时。

如何加快从解耦内存到存储设备的内存表刷新？ 将不可变内存表卸载到解耦内存后，若仍采用原有刷新逻辑，刷新过程可能会变得更加缓慢。直接在解耦内存中的内存表创建 SST 文件并不容易，因为这需要将内存中的数据刷新到持久化存储，同时执行键排序、值序列化、块和索引构建、压缩、元数据管理、预写日志处理和快照控制。由于解耦内存本身不具备刷新执行逻辑，LSM键值存储必须将内存表从解耦内存读回计算节点，才能刷新键值对。此读回过程会引入额外延迟，而且还可能受到同一计算节点上其他前台和后台I/O操作所造成的I/O争用的进一步影响。此外，当解耦内存中大量累积的内存表被刷新到存储设备时，由于 SST 文件的累积，压实缓慢的问题会更加严重（这是LSM键值存储中的一个已知问题，因为每个 SST 文件都需要覆盖整个键空间 L_0）。这会导致写入变慢或写入停滞加剧，并显著影响整体性能。

如何缓解 DM 中内存表搜索缓慢导致的读取性能退化？ 由于键范围重叠，每次读取查询都会访问所有内存表。当在某个内存表中搜索一个键时（例如广泛使用的基于跳表的内存表），需要进行多次内存访问才能最终定位该键值对（或确认未找到）。当内存表维护在 DM 中时，由于通过 RDMA 进行内存访问的速度相对较慢，搜索一个内存表可能导致显著的高延迟。更糟糕的是，一次读取查询需要搜索多个内存表，直到找到该键值对（或在所有内存表中均未找到），这会进一步放大读取延迟。

4 O³-LSM

4.1 架构概览

图 3 展示了 O³-LSM 架构。与其他解耦LSM键值存储一样，O[17, 51], -LSM 实例运行在计算节点上，并将 SST 文件存储在存储设备上。它采用两层写缓冲区，将本地计算节点 DRAM 与大容量解耦内存相结合。新的活动内存表首先缓存在本地内存中，以实现快速插入。随后，它们会作为不可变内存表被卸载到解耦内存中，之后再最终刷新到存储设备。

首先，为解决在 DM 中传输和重建内存表所面临的性能挑战，O³-LSM 在计算节点和 DM 节点上引入了一种新颖的**DM优化内存表**（详见第 4.2 节）。DM优化内存表通过基于指针的索引块（包含跳表的链表节点）和一个键值块重新设计了跳表结构（如果采用分片级优化，键值块将进一步划分为多个键值分片块，详见第 4.4 节）。键值块按顺序存储键值对，跳表索引节点则通过键值块内的偏移量引用这些键值对。因此，无需进行指针重建，即可将键值块直接传输到 DM。

其次，为了加快内存表刷新，我们提出了**协同刷新卸载**（第 4.3 节），将刷写路径与 LSM-KVS 实例解耦，使刷写任务可在远程执行。我们提出了一种刷写卸载协议，用于在内存表所有者与刷写执行器之间传输元数据。由于所有计算节点共享 DM，因此，DM 中的内存表不会先从 DM 读回所属计算节点再进行刷新，而是由选定的任意计算节点或 DM 节点协同刷新。我们引入了协同刷写调度器，用于监控资源信息（即 I/O带宽、CPU利用率）并收集刷写任务元数据（即 DM 节点上的内存表偏移量、原始计算节点）。调度器会使用刷写卸载协议，选择一个有空余 CPU 和 I/O 资源的计算节点或 DM 节点来执行内存表刷新，从而提高资源利用率并加快刷新。

第三，为了进一步提升可扩展性和并行度，并改进 L_0 压实，我们提出了**分片级优化**，根据预定义的键范围将一个内存表划分为多个分片（所有内存表都将采用相同的键范围划分方式）。因此，一个键值块被划分为
多个键值分片块。通过这种方式，一个内存表中的多个键值分片块可以以较高的并行度异步传输，从而进一步提高内存表传输效率。同时，我们不再分别刷新每个内存表，而是将同一LSM键值存储中所有内存表的相同键范围（即合并所有具有相同键范围的键值分片块）一起刷新到 DM 中。基于分片的刷写实际上以细粒度方式结合了刷新和 L_0 压实，显著减少了 L_0 中键范围重叠的 SST 文件数量。因此，O³-LSM 将刷新卸载从粗粒度的内存表级任务细化为细粒度的分片级任务，从而实现更高的可扩展性和并行度，避免 L_0 带来的损失，并显著提升整体性能。

最后，为了缓解通过多次 RDMA 访问在解耦内存中搜索不可变内存表所产生的高开销，O³-LSM 采用了**缓存增强型读取委派**（详见第 4.5 节）。解耦内存中内存表里的热点键值对会被缓存到

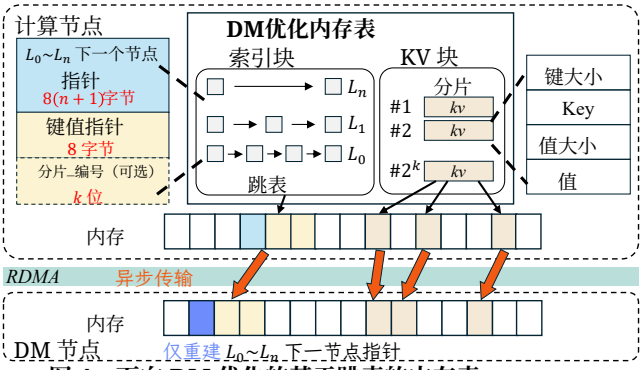


图 4：面向 DM 优化的基于跳表的内存表

计算节点上的小型键偏移缓存中（缓存键及其在解耦内存中的偏移量），然后使用单边 RDMA_READ 获取相应的键值对。如果缓存未命中，O³-LSM 会通过双边 RDMA_SEND 向解耦内存发起读取委派请求，由解耦内存侧的 CPU（读线程）搜索内存表，从而显著减少往返次数。此外，我们还采用了众所周知的 RDMA优化技术（例如门铃批处理和内联），以进一步提升效率。

虽然由于目前的成熟度和可用性，O^3-LSM 在 RDMA 上进行评估 [8, 26, 29, 46], ，但其设计与互连技术无关。即使 CXL 等新兴技术提供了缓存一致、低延迟的访问 [12, 35], ，O^3-LSM 的核心原则仍然至关重要。具体而言，DM友好布局对于缓解代价高昂的远程指针追逐仍然不可或缺，而我们的分片级卸载和读取委派则提供了关键的编排层，以平衡刷写密集型工作负载并最大限度地提高吞吐量。因此，在基于 CXL 的分解式基础设施中，O^3-LSM 仍然适用且有益。

4.2 DM优化内存表

与大多数 LSM键值存储实现类似 [7, 16, 17, 22, 46, 51], ，O³-LSM 使用跳表作为实现内存表的数据结构，因为它具有高效的并发处理能力、快速的点查找和范围查询能力。标准跳表节点在连续内存中存储 k 指针，后接一个键值对。这些指针构成 k 索引层，其中高层指针会跳过一些节点，从而形成稀疏索引。这种结构通常可实现 O(log n) 的查询复杂度。

然而，将当前基于跳表的内存表传输到解耦内存并重建的过程既缓慢又低效。首先，将整个内存表传输到解耦内存，需要在计算节点上遍历所有跳表节点，并在解耦内存中通过指针重建这些节点，这会拖慢内存表卸载过程。其次，在重建过程中，我们需要为跳表节点动态分配解耦内存空间。因此，这些节点会分散在整个解耦内存空间中，造成空间和性能开销。一种直接的解决方案是在解耦内存中预先分配一个足够大的连续内存块（例如 64MB），用于为一个内存表动态分配节点。然而，这种策略可能导致内存碎片和空间浪费。

为解决这些问题，我们提出了一种**DM优化内存表**，如图4所示。我们将内存表分离为一个包含链表节点的基于指针的索引块，以及一个存储所有不含指针的键值对的键值块。索引块中的每个节点保存：1）用于下一节点指针的8（n + 1）字节，2）一个键值

指针，占8字节，指向键值块中对应键值对的偏移量，以及3）一个具有 k 位的分片_id（如果应用了分片级优化）。在键值块中，键值对按顺序追加到一个连续内存块中，并组织为元组（<键、值、偏移量>）。这种分离使内存表能够存储在不同的、更小的连续内存块中。键值块可以直接传输并存储到解耦内存中预先分配的连续内存块，而无需进一步处理。仅需对索引块进行重建，以使用键值对的新内存地址修正指针。

内存表变为不可变后，我们会将其加入异步传输队列。后台工作线程首先将索引块序列化为紧凑、连续的缓冲区，将此缓冲区注册为单个远程直接内存访问内存区域，然后向解耦内存上预先分配的区域发起单边远程直接内存访问_WRITE。键值对无需序列化，因为它们已作为字符串以长度前缀编码在键值块中。工作线程注册键值块，并通过单边远程直接内存访问_WRITE 将其传输到解耦内存中指定的区域。传输过程中，我们会维护一条小型元数据记录（总计 114 字节），其中包含索引块和键值块在计算节点和解耦内存上的起始地址。对于 64 MB 的内存表，这条紧凑记录可确保元数据开销低于 0.0002%。写入完成后，解耦内存端会通过刷新跳表节点的指针来重建索引块。随后，内存表会在解耦内存上标记为可搜索状态，传输作业也随之完成。

请注意，存储在索引块中的每个键值对的偏移信息无需在解耦内存中更新。O^3-LSM 会在查询时，通过添加键值块在解耦内存中的适当内存偏移量，来修正索引中的这些字段。此外，由于所有键值对都采用长度前缀编码并按顺序追加，因此键值块无需在解耦内存中更新或重构。例如，假设计算节点上某个键值块的起始内存地址为 A。键值块传输到解耦内存后，其起始内存地址为 B。键值块中某个键值对的偏移量为 C，该偏移量存储在索引块的一个索引节点中。因此，将键值块内的偏移量与键值块在解耦内存中的地址偏移量（即 $C + B - A$）相加，即可确定该键值对在解耦内存中的正确地址；这一修正可在搜索内存表时以 O(1) 的时间复杂度完成。

4.3 协同刷新卸载

在传统的 LSM键值存储 设计中，刷新过程与创建刷新作业的 LSM-KVS 实例相耦合。当使用解耦内存作为外部写缓冲区时，这种耦合会带来两个缺点：（1）由于解耦内存缺少刷新执行逻辑（例如键排序和刷新元数据管理），发起刷新的 LSM-KVS 必须从解耦内存取回不可变内存表，再将其刷新到存储设备，从而产生传输和（反）序列化开销；（2）内存表刷新仍可能与前台读取和后台压实争用网络 I/O。随着解耦内存中积累的内存表越来越多，这些开销也会增加，并可能限制前台读写。

为应对这些挑战，我们提出了一种新颖的**协同刷新卸载**（图5）。我们将刷新控制平面与LSM-KVS实例解耦，使其能够在任何具备解耦内存访问能力的节点上远程执行。该设计包含两个组件：1）一种轻量级的刷新卸载协议，用于准备、协调并传递所有必需的元数据，编排工作流和故障处理，并最终确定作业状态；以及2）一种逻辑上集中式的刷新调度器，根据负载和局部性将刷新任务分配给计算节点或解耦内存节点。如图5所示，O^3-LSM分四个阶段卸载刷新，支持三种执行模式（本地、解耦内存内和远程计算节点），并使用六种控制消息。

阶段 1：准备。调用内存表刷新时，所属的 LSM-KVS 实例会在细粒度锁的保护下对其数据库状态进行快照，记录待刷新的内存表的序列号边界，并确保预写日志已持久化至该序列号，然后再卸载刷新任务。接着，所属的 LSM-KVS 实例会构建一个紧凑的刷盘元数据包（总计 572 字节），其中包含选项（例如压缩和块设置）、文件元数据（例如目标层级、文件号）、键值存储状态元数据（例如列族标识和序列水位）以及预写日志状态（例如日志文件标识和已持久化偏移）。该数据包还会编码索引块和键值分片块在解耦内存中的位置。由于 LSM-KVS 中许多元数据对象是共享的，我们会预先计算一张小型依赖图，并在打包时按固定顺序遍历，以消除重复项。最后，所属的 LSM-KVS 实例会向调度器发送一条携带刷盘元数据包的 PREPARE消息。

阶段 2：分配。收到 PREPARE消息后，调度器将刷写任务推入先进先出队列，并按照到达顺序分派作业，以确保公平性。对于每个作业，调度器会查看当前活跃的执行器池并选择一种执行模式：本地 （所属 LSM键值存储执行刷新）、解耦内存内 （由 DM 节点就地执行，无需将数据读回计算节点），或远程计算节点 （由另一计算节点从解耦内存读取内存表并执行刷写任务）。选择目标节点后，调度器向选定的执行器发送 ASSIGN消息并启动超时计时。执行器会回复 ACCEPT；如果发生超时，调度器则将作业重新分配给另一节点上符合条件的执行器。收到 ACCEPT 后，调度器传输刷盘元数据包，并等待执行器发送 COMMIT 或 ABORT。

阶段 3：执行。执行器收到刷盘元数据包后，会使用单边远程直接内存访问_READ从解耦内存获取所引用的内存表数据。执行器无需进行指针重建即可直接读取键值块，并通过基地址重定位重新锚定最小化的内存索引。执行器（刷盘线程）按照序列号顺序合并记录，构建所需的索引和过滤器块，根据配置进行压缩，并按照刷盘元数据的指定将 L_0 SST 文件写入存储设备。如果在完成前发生故障，执行器会中止作业并向调度器发送 ABORT消息，以便安全地重新分配该作业（尚未发布任何状态）。当刷新输出在存储设备中持久化后，执行器会向调度器发送 COMMIT消息，并附上生成的文件元数据，以便所属 LSM键值存储更新刷新状态。

阶段 4：最终确定。收到 COMMIT 后，调度器将生成的文件元数据转发给所属的 LSM 键值存储。所属的 LSM 键值存储验证元数据，在 Manifest 中安装新的数据库状态，并在数据库互斥锁的保护下从其元数据中移除内存表，然后通知解耦内存回收空间，确保在途读取要么能找到数据，要么回退到 DS 读取。随后，所属的 LSM 键值存储向调度器发送确认。调度器将作业标记为完成，将其从队列中移除，并

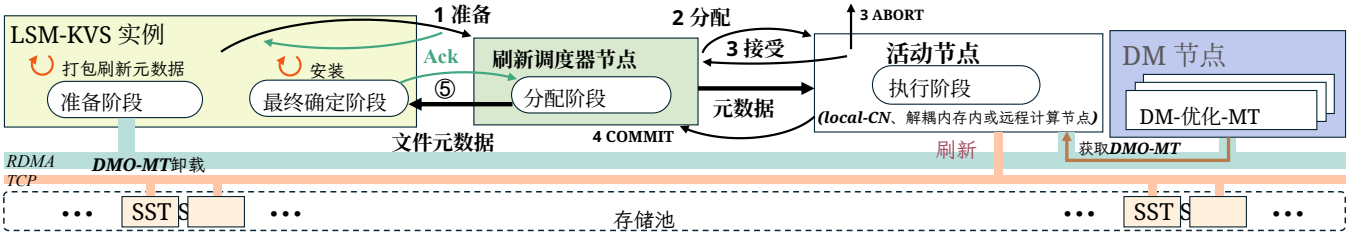


图 5：协同刷新卸载

释放其跟踪状态和计时器。此阶段可安全地发布结果，并及时释放资源。

刷新调度器。 调度器在计算节点和 DM 节点之间维护一个动态的刷写执行器池，以平衡刷新负载并缓解某些计算节点上发生的写入突发。调度器使用心跳信号进行节点注册和存活监测。为了平衡卸载的刷新任务，我们采用确定性的成本模型，为每个节点 i 分配一个负载因子 $\text{load}_i = w_{\text{cpu}} u_i^{\text{cpu}} + w_{\text{io}} u_i^{\text{io}} + w_{\text{queue}} u_i^{\text{queue}}$。其中，权重（$w$）是预先配置的缩放系数，用于根据硬件瓶颈优先分配资源（例如，在 I/O 受限环境中设置较高的 w_{io}）。利用率指标（u）是经过归一化的值：u_i^{cpu} 表示活动刷写线程数与线程池总大小的比值，u_i^{io}表示当前存储写入吞吐量相对于节点峰值校准带宽的比例，而 u_i^{queue} 则衡量当前待处理分片数量相对于最大允许队列深度的比例。

基于这些信号，调度器使用公式 $\text{cost}_i = \text{字节} \cdot (1 + \text{负载}_i)$，计算将大小为 字节 的刷写任务分配给节点 i 的预估成本。我们采用贪心策略，选择成本最低的节点来执行刷写任务。例如，如果节点 A 的负载较低，为 0.1，而节点 B 的负载高达 0.9，那么一个 64 MB 的分片对应的成本分别为 70.4 和 121.6，因此调度器会选择节点 A。为确保稳健性，调度器会在每个分片完成后，根据最新的遥测信号重新计算负载$_i$，避免瞬时误判不断累积。为提升可扩展性，可以部署多个调度器实例；更复杂的调度策略则留待未来研究。

4.4 分片级优化

通过单次远程直接内存访问_WRITE 发送一个 64 MB 的块，可能导致突发性的带宽争用，尤其是在同时存在其他远程直接内存访问流量[32]时。传输如此大的内存表也会限制并行度，且无法解决压实缓慢的问题 L_0。为此，我们提出将内存表划分为多个互不重叠的键范围（称为分片）。这样，单个键值块便被拆分为多个连续的键值分片块。同时，我们提出并行刷新多个分片块，

将来自不同内存表的相同键范围一起处理，实际上将 L_0 压实与刷新结合起来。这可以提高刷新的并行度，并解决 L_0 压实带来的性能损失。

内存表分片。 我们使用每个键的前 k 位，将内存表键空间划分为2^k 个互不重叠的键范围（分片）。在每个分片内，键值对会追加到一段连续区域中，形成一个 键值分片块。每个索引块节点除了键值指针，还会存储一个 k 位的分片_id；键值对则通过其在键值分片块中的偏移量寻址。一旦

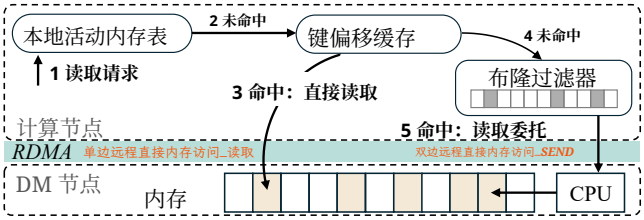


图6：缓存增强型读取委派的缓存增强型读取委派。

分片块变为不可变后，就会独立传输到解耦内存，因此我们无需等待整个内存表封存，从而缩短等待时间并提高并发度。采用这种设计，较小的键值分片块可以异步并行传输，无需进行指针重建，从而充分利用 RDMA 带宽。在解耦内存上进行指针校正时，我们使用键值分片块的起始地址，而不是整个键值块的起始地址，来进行基地址重定位。

分片级刷新卸载。为进一步提升可扩展性和并行度，并提高 L_0 压实效率，O³-LSM采用分片级刷新。O³-LSM会物化分片级刷新元数据（例如，分片 ID 和解耦内存上键值分片块的偏移量），并重新设计刷新流程，将解耦内存中来自不同内存表、属于相同键范围的多个键值分片块聚合为一个刷写任务。这将粗粒度的内存表级刷新转变为细粒度的分片级操作，使刷新和 L_0 压实能够同时执行。与传统设计在累积的内存表达到大小限制时触发刷新不同，只有当不可变内存表中某个分片的键值分片块总大小超过一个内存表大小限制（例如 64 MB）时，才会触发分片级刷新。这样生成的 L_0 SST 文件大小大致与内存表相同，且键范围重叠极少，从而减轻 L_0 压实带来的性能损失。分片级优化还能处理工作负载倾斜。调度器通过将热点分片分配给最合适的节点，实现细粒度控制，在整个集群内平衡 CPU 和网络负载。

4.5 缓存增强型读取委派

在原始 LSM键值存储设计中，响应读取查询（例如 *Get* 和 *Scan*）可能需要遍历一个或多个内存表。在解耦内存中的内存表上执行多次单边远程直接内存访问_READ需要：1）解引用以获取键；2）根据键比较结果解引用以获取下一个节点地址；3）重复这一过程，直至找到键值对（或返回未找到）。这可能会使读取性能降低一个数量级甚至更多 [49]。

为解决这一多轮遍历问题，可以考虑两种主要的优化方法：1）将数据从

DM 读取回计算节点，并在计算节点上进行本地搜索（本地处理速度更快，但搜索远程索引块可能较慢）；或者 2）将搜索任务卸载到远程 DM（可在 DM 上异步进行本地搜索，但计算资源可能有限）。我们结合这两种方法，提出一种适用于本地 DRAM 和远程 DM 的**缓存增强型读取委派**。如图 6 所示，对于频繁访问的键值对，我们在计算节点上的 O³-LSM 中实现本地键偏移缓存，从而减少内存表查找所需的往返次数。对于访问频率较低或新近访问的键值对，我们采用读取委派方法，利用 DM 的计算能力进行 DM本地查找，从而减少搜索索引块所需的多次 RDMA 往返。这种方法在多层架构中有效结合了单边和双边 RDMA 操作的优势。

本地键偏移缓存。 我们为 *O*³-LSM 设计了一种小型本地键偏移缓存。它采用基于LRU的策略缓存热点条目，将键映射到其对应的 DM 内存偏移量。为尽量减少计算节点的内存占用，我们缓存偏移量而非完整的键值对值；缓存命中时，只需通过一次单边远程直接内存访问_READ 即可获取值。键偏移条目通过以下两种方式插入：1) 当活动内存表缓存于本地 DRAM 中时，O³-LSM 会跟踪每个键值对的读取访问频率，并用 3 位的 Freq 表示，从而支持 8 个不同的频率等级。在传输过程中遍历内存表时，会为达到预设频率阈值（例如 4 或更高）的热点键值对创建键偏移条目，并将其插入缓存。借助这种方法，单边远程直接内存访问_READ 可直接使用缓存的 DM 偏移量获取目标键值对，而无需搜索索引块。2) 当本地键偏移缓存查找未命中时，将通过读取委派从 DM 检索该键值对。随后创建并缓存相应的键偏移条目，以加快对同一键值对的后续请求。此外，O³-LSM 会为存储在 DM 中的内存表的每个键值分片块创建并缓存一个布隆过滤器。该过滤器用于判断是否需要读取委派，从而有效减少平均远程访问次数。

在解耦内存（DM）上进行读取委派。如果本地键偏移缓存未命中，我们会利用 DM 的计算资源来委派搜索任务。O³-LSM 首先根据查找_键的分片_标识符，遍历相应键值分片块的布隆过滤器。如果任一布隆过滤器返回肯定结果，O³-LSM 就会将查找_键和内存表标识符集合打包成读取委派请求。然后，O³-LSM 使用双边远程直接内存访问_SEND 向 DM 发起远程过程调用。在 DM 端，我们维护一个包含多个轮询线程的工作线程池，用于接收和处理这些读取委派请求。收到来自计算节点的请求后，工作线程会轮询并根据请求在目标内存表上执行本地查找（委派的 *Get* 操作），并传回结果，直至找到键值对（若所有内存表中均不存在，则返回未找到）。O³-LSM 收到来自 DM 的结果后，会将该键及其远程偏移量作为一个键偏移条目缓存在本地键偏移缓存中。

其他优化。我们提出了几项 RDMA 优化：1）**门铃批处理**。我们每次提交 8–16 个单边 RDMA_READ，并只敲响一次网卡门铃，从而分摊初始化开销。2）**针对委托回复的自适应内联**。对于委托查找，DM 节点会直接在 RDMA 回复中返回较小的值。对于较大的值，它会返回元数据，以便执行一次后续单边读，从而有效地将往返次数降至最低。

4.6 容错

*O*³-LSM 将计算节点的预写日志视为权威状态，并将 Manifest 作为唯一的可见性边界，以此确保一致性。**作业级故障**。 这些是影响单个操作的暂时性错误：

1）内存表卸载：如果卸载到解耦内存失败，计算节点会释放已分配的解耦内存区域，并根据预写日志在本地重建内存表。未完成的远程内存表绝不会被注册。2）分片级Flush：如果任务在执行刷新期间失败，调度器会丢弃部分生成的 SSTable，并使用解耦内存中的不可变内存表重新调度该分片。只有在 Manifest 更新成功后，数据才会变为可见。3）读委托故障：如果对解耦内存的委托读失败或超时，计算节点会重试请求；如果数据已持久化，则直接从 DS 层读取。这样可确保暂时性的解耦内存或网络问题不会导致请求失败。

节点级故障。 这类故障涉及整个系统组件崩溃：1）计算节点故障：重启后，计算节点会重放本地预写日志，以重建内存表，并将其重新卸载到解耦内存。调度器会清除与故障计算节点关联的过期解耦内存状态。2）解耦内存故障：调度器会中止所有涉及故障解耦内存的活动作业。计算节点会根据预写日志在本地重建已卸载的内存表。3）调度器故障：调度器会从持久化日志中恢复状态，并选择继续执行已记录的 COMMIT，或丢弃未完成的作业并要求计算节点重试。 *O*³-LSM 在 Manifest 边界上维持全有或全无保证。故障作业或节点产生的部分输出可能需要重试，但绝不会导致 SSTable 被部分安装或元数据状态不一致。

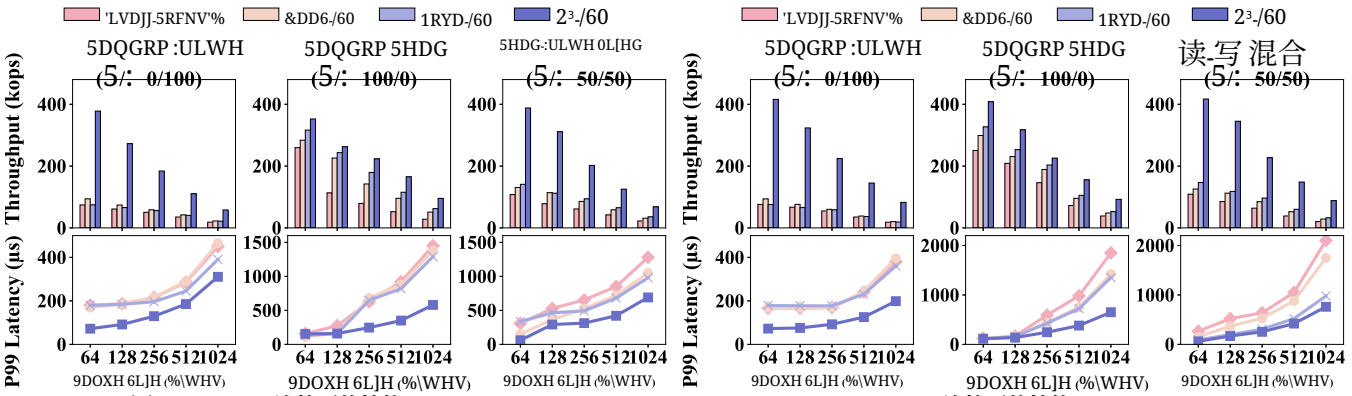
5 评估

我们基于 RocksDB v8.2.0 和 Disaggregated RocksDB 实现了 O³-LSM。源代码可在 GitHub [38] 上获取。我们进行了全面评估，以回答以下问题：1）与最先进的解耦LSM键值存储相比，O³-LSM 是否能显著提升性能？2）各项主要设计的性能表现如何？3）在实际应用场景下，性能和可扩展性如何?

5.1 实验设置

硬件平台。我们在 CloudLab 的[19] c6220 实例上进行评估，每个实例配备 2 × Xeon E5-2650v2 CPU（每个 8 个核心）、64 吉字节内存和 1 TB 硬盘。为模拟解耦式配置，我们将计算节点配置为使用所有可用核心，但仅使用 2 吉字节内存；而解耦内存节点则配置为使用 4 个核心和 64 吉字节内存。所有计算节点和解耦内存节点均通过 Mellanox FDR ConnectX-3 网络接口卡连接（连接速率为 40 Gbps）。我们在存储节点上部署 HDFS，将其作为基于 TCP 的 DS（采用与 CaaS-LSM 相同的 DS 配置）。

基线系统。 为展示我们设计的有效性，我们采用针对 DS [17] 优化的 RocksDB 作为基线系统（**Disagg-RocksDB**）。我们还纳入了 **CaaS-LSM** [51]（一种采用 DS 远程压实的先进 LSM）和 **Nova-LSM** [25]（一种通过远程直接内存访问实现 DS访问的解耦LSM键值存储）。这三种基线系统的配置均最多使用 8



（a）1 Gbps DS连接下的性能　　(b) 2.5 Gbps DS连接下的性能

Figure 7：1 Gbps 和 2.5 Gbps DS连接下的微基准测试性能对比

计算节点上的本地内存表（总计 512 MB）。相比之下，我们的设计 **O³-LSM** 仅使用 2 个本地内存表（128 MB），并在远程DM节点上最多使用 6 个内存表。此外，我们还评估了 **dLSM** [46]（一种基于 RDMA 的 DM 上完全驻留内存的 LSM键值存储），并将其作为性能上限基线（即它不具备持久化能力，预期能在基于 RDMA 的 DM 上实现最佳性能）。

工作负载。 我们使用 db_bench 和 **YCSB** [13] 评估 O³-LSM。对于 db_bench，我们采用 随机填充、随机读取和 随机读写 基准测试。对于 YCSB，我们采用具有 4 种读写比例和 2 种键值分布（均匀分布和齐夫分布）的工作负载。在齐夫分布中，键按齐夫分布选取，偏斜度设为 0.99。除非另有说明，所有实验均使用 1 个 DM 节点、1 个计算节点和 3 个存储节点。可扩展性和性能分解评估将使用更多计算节点。O³-LSM 和其他基线系统采用相同的配置。对于内存组件，内存表大小设为 64 MB，块缓存大小为 512 MB，本地键偏移缓存大小为 64 MB。最大_后台_作业 数量设为 4，用于压实和刷新。降低_写入_触发阈值 设为 L_0 32，停止_写入_触发阈值 设为 48。调度权重 w_{cpu}、 w_{io} 和 w_{queue} 均设为 1.0。对于 NovaLSM，我们将内存表_分区_数量 配置为 12，以最大限度提高并行度并充分利用 RDMA 带宽。每项基准测试执行 5000 万次操作，键和值的大小分别设为 16 字节和 64 字节。所有评估均执行 3 次，并报告平均值。

5.2 整体性能评估

我们使用 db_bench 微基准测试，在 DS 访问的网络 I/O 受限（1 Gbps）和充足（2.5 Gbps）两种情况下，将 O³-LSM 与三个基线系统进行整体性能比较，如图7a 和图 7b 所示。此外，我们还通过 YCSB 微基准测试评估 O³-LSM 在不同读写比例、键值分布以及范围查询下的性能。

写入性能。 我们首先关注不同值大小下的高强度随机写入基准测试。我们使用每秒操作数（Ops/s）来衡量吞吐量。如图 7a 所示，在 1 Gbps 网络带宽下，O³-LSM 的吞吐量最高可比 *Disagg-RocksDB*提升 4.1*X*，比 *CaaS-LSM*提升 3.2*X*，比 *Nova-LSM*提升 3.9*X*（分别在总计使用内存表时提升高达 6.4*X*、

4 6*X*　　5 6*X*　　96　　. 和 .）。

P99延迟也降低了约 32% 至 59%，这表明 O³-LSM 的写入性能有显著提升。*Disagg-RocksDB* 会因值大小增加而出现吞吐量下降，这是因为写入压力增大，导致发往 DS 的网络 I/O 负载加重，并增加了 L_0 压实开销。类似地，随着值大小增加，O³-LSM 相较于 *CaaS-LSM* 的吞吐量提升更为明显。这是因为值大小增加时，O³-LSM 的分片级刷新卸载 可降低写放大并实现更细粒度的调度，从而提升整体性能。对于 *Nova-LSM*，吞吐量会随着值大小增加而下降，同时 P99延迟急剧上升。尽管使用远程直接内存访问连接计算集群和存储集群，但通过网络管理较大数据块的开销最终超过了其分片架构所带来的带宽优势。值得注意的是，尽管使用速度慢得多的基于 TCP 的 DS，O³-LSM 在吞吐量和 P99延迟方面均优于 *Nova-LSM*。

图 7b 展示了带宽充足（DS连接为 2.5 Gbps）时的随机写入性能评估。与网络带宽受限于 1 Gbps 时相比，所有四种解耦式 LSM 键值存储的性能均有所提升。随着值大小增加，O³-LSM 的吞吐量相较于 *Disagg-RocksDB*、*CaaS-LSM* 和 *Nova-LSM* 分别最高提升 4.5*X*、3.4*X* 和 4.4*X*（总计使用 96 个内存表时，最高提升 6.9*X*、5.7*X* 和 6.2*X*）。此外，将计算节点到存储设备的网络带宽从 1 Gbps 提高到 2.5 Gbps 后，O³-LSM 的 P99延迟降低约 37% 至 60%，表明 O³-LSM 在带宽充足时能够实现更好的性能。作为高吞吐量、非持久化的 LSM 参考系统，dLSM 在值大小为 64B 时达到 1.23 百万次操作（Mops），为使图表清晰，故未在图中展示。值大小从 64B 增至 1024B 时，由于固定的网络带宽与有效载荷大小直接制约彼此，所有基线系统的操作数均有所下降；同样，dLSM 的吞吐量也从 1.23 降至 0.58 Mops。以该参考系统在 64B 时的表现为基准，O³-LSM 达到 0.415 Mops（为 dLSM 的 34%），而其他持久化 LSM 键值存储基线系统的最高表现仅为 0.09 Mops（为 dLSM 的 7%）。

读取性能。 我们使用 db_bench 中的随机读取 基准测试评估点查找性能。首先，我们加载了 5000 万条记录，并等待所有压实完成后，再进行随机读取 测试。如图 7a 和图 7b 所示，与带宽受限时相比，在带宽充足时，所有系统的读取吞吐量都略有提升。对于 P99 延迟，在带宽充足且值较大时，*CaaS-LSM* 和*Nova-LSM* 的延迟分别显著降低了

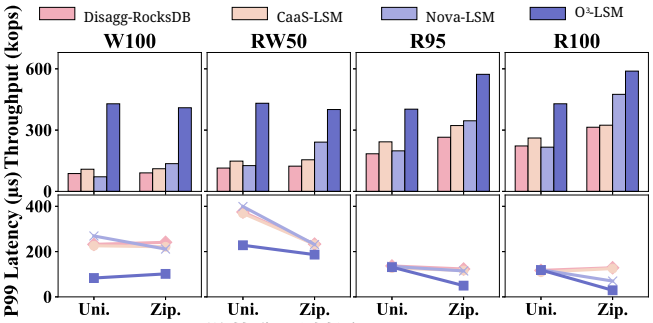


图 8：YCSB 微基准测试性能。

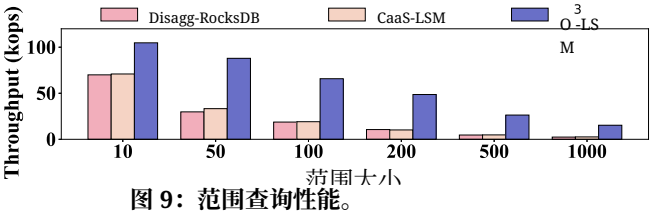


图 9：范围查询性能。

35% 和 33%。在带宽受限时，与其他基线系统相比，O³-LSM 的 P99 延迟最多降低了 69%。与*Disagg-RocksDB*、*CaaS-LSM*和*Nova-LSM*相比，O³-LSM 的读取吞吐量分别最高提升了 1.8*X*、0.6*X* 和 0.3*X*（总共使用 96 个内存表时，最高分别提升 2.2*X*、1.8*X* 和 1.5*X* ）。然而，随着值大小增加，所有系统的性能都会下降，O³-LSM 尤为明显。这是因为，尽管缓存增强型读取委派能够加速对存储在解耦内存（DM）上的内存表的读取，但较大的值会消耗更多远程直接内存访问带宽，导致性能下降。与 dLSM 相比，O³-LSM 达到了理论上限的 11%（64B 时为 2.24 百万次操作，1024B 时为 1.25 百万次操作）。其他基线系统仅能达到 0.2 百万次操作（dLSM 的 8.9%）。总体而言，缓存增强型读取委派有效降低了 O³-LSM 在解耦内存（DM）上命中内存表时的远程内存访问开销。更重要的是，分片级刷新卸载生成的 L_0 SSTable 重叠键范围少得多。由于大部分数据位于存储设备（DS）上，大多数读取查询都由其处理。因此，对于点查询，O³-LSM 将查找从探测多个键范围重叠的文件转变为更少次数的探测；对于扫描，则降低了迭代器扇入。这种结构上的改进降低了读放大，消除了冗余的布隆过滤器或元数据检查，同时减少了 CPU 和 I/O 开销。因此，通过最大限度减少 DS 读取，该设计实现了优于基线系统的读取性能。

读写混合负载。 我们使用*readrandomwriteran-dom*基准测试，在不同带宽压力下评估 50:50 的读写比例。如图 7a 和图 7b 所示，O³-LSM 在两种情况下都取得了最佳性能。与*Disagg-RocksDB*、*CaaS-LSM*和*Nova-LSM*相比，O³-LSM 的总体吞吐量分别最高提升 3*X*、2 3*X* 1 9*X* ，以及 。（总计使用 96 个内存表时最高提升 3.9*X* 、 3.6*X* 、 3*X* ），并将 P99 延迟最高降低 76%。O³-LSM 的吞吐量最高可达 dLSM 的 16.5 %（64B 时为 1.53 百万次操作，1024B 时为 0.82 百万次操作），而其他基线系统的吞吐量均低于 dLSM 的 10%。

工作负载	系统	L0 压缩	L1 压缩	L2 压缩	总计
随机写入 (50M, 256B)	O³-LSM	12.6	18.4	19.1	50.1
	Disagg-RocksDB	12.7	22.4	24.2	59.3
混合 (50/50) (50M, 256B)	O³-LSM	6.3	9.2	16.4	31.9
	Disagg-RocksDB	6.4	12.1	21.3	39.8

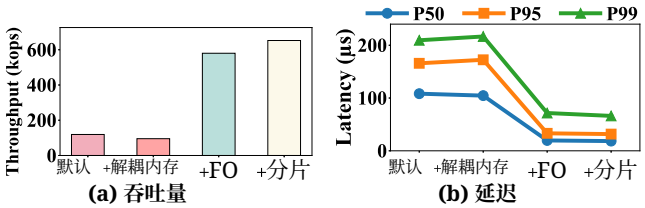
表 2：写放大分析（写入数据量，单位：吉字节）。

微基准测试。 我们在此评估中使用了广泛采用的 YCSB 基准测试。我们采用均匀分布 和齐夫分布 键分布，并评估了 4 种不同的读写比例。我们重点关注四种工作负载：1）**RW50**（50 % 读取、50% 写入），2）**R95**（95% 读取、5% 写入），3）**R100**（100% 读取），4）**W100**（100% 写入），值大小为 64 字节，带宽为充足的 2.5Gbps。图 8 显示，O³-LSM 在所有评估中都取得了最佳性能。我们得出两个主要结论：第一，与均匀分布 相比，O³-LSM 的优化在齐夫分布 下更有效，尤其是在读密集型工作负载（R95 和 R100）中。齐夫分布 会产生热点键值对，键偏移缓存可以将其有效缓存在计算节点本地内存中，从而显著降低访问存储设备的延迟，相较于其他三个基线系统尤为明显。第二，在齐夫分布 下，O³-LSM 在写密集型工作负载（W100）中的性能略有下降。倾斜分布使大多数插入集中在少量分片中，影响分片级并行度，并导致出现大量键范围重叠的 SST 文件于 L_0。

范围查询。 我们使用不同的范围大小评估范围查询的性能。如图 9 所示，O³-LSM 的性能优于 *Disagg-RocksDB* 和 *CaaS-LSM*。随着范围大小增加（例如从 10 到 1000），三个系统的吞吐量均下降。不同的是，O³-LSM 实现了显著更好的吞吐量提升，相对于 *Disagg-RocksDB* 和 *CaaS-LSM* 分别最高可达 5.2*X* 和 4.6*X*。这一改进源于两个关键因素：1）O³-LSM 执行分片级Flush，通过将许多连续键分配到同一分片，有利于范围查询，从而在从存储设备读取 SST 文件时显著减少预取时间。2）O³-LSM 节省本地内存以支持更大的块缓存，从而进一步减少 DS 输入/输出。

5.3 性能细分分析

写入性能。 如图 10a 所示，仅启用解耦内存（**+解耦内存**）来扩展写缓冲区，并不能缓解写缓冲区受限导致的写入变慢问题。事实上，与 *Disagg-RocksDB*（**默认**）相比，其吞吐量下降约 18%，延迟也略有增加，这是由于内存表传输/重建速度较慢，且显式刷写操作会引入延迟。当同时在解耦内存中刷新更多内存表时，性能甚至会更差。当待刷新的内存表数量超过 10 个时，在解耦内存中刷新内存表的速度比原始 *Disagg-RocksDB* 更慢。然而，集成所提出的 刷新卸载 （**+FO**）后，刷新吞吐量提高了 34.6*X*，从 5MB/s 提升至 178MB/s（内存表上限为 100 个）。更重要的是，协同刷新卸载具有很高的可扩展性（即随着内存表数量增加，聚合刷写吞吐量也几乎线性增长）。此外，将刷新卸载 拆分为分片级操作（**+分片**）后，我们实现了更高的整体写入吞吐量（652K 操作数/秒）和更高的刷新吞吐量（比 **+FO** 提高 15%）。



(a) 吞吐量　(b) 延迟

图 10：随机填充 工作负载上的消融研究。

方法	MP 刷写		IF FM/其他		总计（毫秒）
Disagg-RocksDB	3.2	877	2.5	N/A	882.7
CaaS-LSM	3.1	874	2.6	N/A	879.7
Nova-LSM	1.9	852	2.1	N/A	856.0
朴素 DM 方案	3.2	877	2.5	286.0†	1168.7
O³-LSM	**4.7**	**875**	**3.2**	**54.0**	**936.9**

† 包括朴素实现中的传输与重建（218 ms）以及检索（68ms）。**表 3：刷新延迟分解分析。**

写放大分析。 表 2 列出了不同 LSM 层级压实过程中写入数据量的细分情况。 *O*³- LSM 的总写放大（WA）始终低于 Disagg-RocksDB。具体而言，对于 5000 万次随机写入工作负载， *O*³-LSM 将写入的数据总量减少了约 15.5%。这一减少主要体现在 L_1 和 L_2 压实过程中。通过采用分片级刷新卸载和内存表卸载， *O*³-LSM 生成的重叠 L_0 SST 文件更少，从而显著减少后续压实层级中的重叠键范围，进而降低整体写放大。

刷新性能分解。如表 3 所示，与直接将内存表从本地内存刷新到 DS（本地基线）相比，O³-LSM 的分片级刷新卸载仅使平均端到端延迟增加 7%。在整体时间线上，将内存表传输到解耦内存占总耗时的 18.7%，但我们采用异步内存表传输设计，通过将传输与执行重叠，使这部分开销脱离刷盘关键路径。在刷新卸载关键路径中，刷盘元数据打包/解析（**MP**）平均耗时 4.7ms，安装/收尾（**IF**）平均耗时 3.2ms，而执行器获取元数据和内存表块（**FM**）占总执行时间（54ms）的 5.7%；其余时间用于合并/序列化/写入 DS（**刷新**）。这些结果表明，所提出的协同刷盘卸载协议开销不大，且大部分被重叠操作掩盖；主要开销来自数据移动和 DS I/O。

读取性能。 如表 4 所示，我们将 O³-LSM 与几个基线系统进行比较，以凸显其读取性能优化的有效性。首先，我们介绍一种直接的方法：O³-LSM 使用多次单边 RDMA_READ 操作来搜索解耦内存（DM）中的内存表（**单向读取**）。单边读只能达到 8,426 操作数/秒，并导致极高的延迟（547微秒），甚至显著差于*Disagg-RocksDB*。这表明，直接使用单边远程直接内存访问_READ操作在解耦内存处搜索内存表会导致极高的开销。其次，当我们在O³-LSM中启用读取委派 （**RD**）时，其性能优于*Disagg-RocksDB*，在均匀分布和齐夫分布下，吞吐量分别提高0.23倍和0.27倍，P50延迟分别降低80%和85%。最后，通过增加本地键偏移

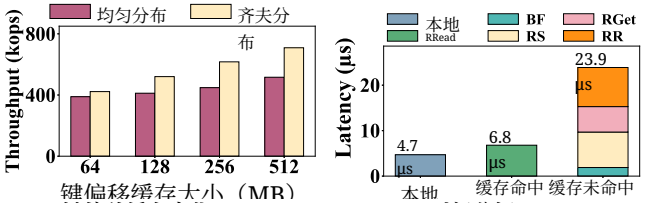
	P50延迟 (µs)		吞吐量（千操作每秒）	
基线系统	**均匀分布**	**齐夫分布**	**均匀分布**	**齐夫分布**
单向读取	547.369	501.327	8.426	11.265
Disagg-RocksDB	37.465	31.129	296.975	325.315
RD	5.425	6.941	366.975	414.791
RD+LC	5.370	5.597	424.460	559.711

表4：读性能分解（均匀分布与齐夫分布）。

工作负载 / 系统	计算节点—解耦内存	解耦内存—计算节点	计算节点—存储设备	解耦内存—存储设备
随机读取（*O³-LSM*）	1.1	6.7	4.2	0.0
随机读取（*Disagg-RocksDB*）	N/A	N/A	10.7	N/A
随机写入（*O³-LSM*）	12.6	8.4	46.1‡	4.3
随机写入（*Disagg-RocksDB*）	N/A	N/A	59.3	N/A
混合（*O³-LSM*）	6.3	5.2	29.6‡	2.4
混合（*Disagg-RocksDB*）	N/A	N/A	39.8	N/A

‡流量分布在两个计算节点之间：**随机写入** (25.36/20.74 吉字节)，**混合** (19.5.2/10.1 吉字节)。

表 5：网络 I/O 分解（数据量，单位：吉字节）。



(a) 键偏移缓存变化　(b) 时间分解

图11：读取性能分析。

缓存（**RD+LC**）作为完整的缓存增强型读取委派设计，与仅使用RD的方案相比，**RD+LC**在均匀分布和齐夫分布下，吞吐量分别提升15%和35%，延迟分别降低2%和19%。

键偏移缓存大小。 为了分析键偏移缓存大小对读取性能的影响，我们使用均匀分布和齐夫分布随机读取 工作负载进行了实验。如图11a所示，随着键偏移缓存容量的增加，O³-LSM 的性能有所提升。具体而言，对于齐夫分布工作负载，将缓存大小从 64 MB 增加到 512 MB 后，吞吐量提升了 67%。相比之下，对于均匀分布工作负载，吞吐量提升了 32%。这表明，我们的设计能够借助更大的键偏移缓存显著提升读取性能，并且在偏斜工作负载下效果更佳。

读取时间分解。 如图11b所示，我们展示了三种读取情形。第一种方案在本地内存中查找内存表（**本地**），耗时 4.7us。第二种和第三种对应于我们的缓存增强型读取委派。缓存命中时（**缓存命中**），一次单边 RDMA_READ 即可在 6.8us 内完成（**RRead**）。缓存未命中时（**缓存未命中**），流水线会先执行布隆过滤器检查（**BF**，1.89us），然后通过 RDMA_SEND 向解耦内存节点发送请求（**RS**，7.8us），接着在解耦内存上进行远程搜索（**RGet**，5.6us，由于额外的拷贝，耗时高于本地搜索），最后通过 RDMA_RECV 接收结果（**RR**，8.6us），总耗时为 23.9us。两次双边 RDMA 操作均通过门铃批处理进行摊销，从而降低了有效开销。与耗时 29us 的朴素方案相比，未命中路径将延迟降低了 17.6%（5.1 us），缓存命中路径则将延迟降低了 76.6%（22.2us）。

不同最大内存表数对性能的影响。图 12 展示了允许在 O-LSM 中使用的最大内存表数不同情况下的性能

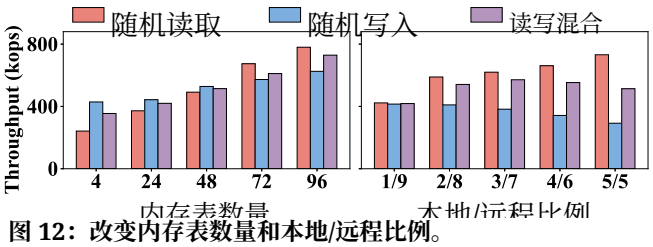


图 12：改变内存表数量和本地/远程比例。

系统	指标	计算节点→解耦内存	解耦内存→计算节点	计算节点→存储设备	解耦内存→存储设备
O^3-LSM（含分片）	Peak	146.2	86.7	265.0	42.3
	平均值	58.4	34.2	234.0	40.2
O^3-LSM（无分片）	Peak	963.2	216.5	297.0	259.0
	平均值	39.2	24.6	214.0	36.8

表 6：网络流量的分片级优化（MB/s）。

（计算节点中最多允许 2 个，其余将转移到解耦内存）。由于解耦内存中可以存储更多内存表，O³-LSM 在不同工作负载下均表现出更好的性能。这一结果与我们的初衷一致：在写缓冲区中缓存更多内存表可以提升整体性能。这一结果彰显了我们的设计通过使用解耦内存扩展写缓冲区，成功应对了新的挑战。对于写缓冲区中固定的最大内存表数（在我们的测试中设为 10 个），本地内存表与远程内存表的比例也会影响整体性能。如图 12 所示，提高本地内存表的占比可以提升读取性能，因为能够在本地内存表中搜索更多键值对。然而，写入性能会下降，因为我们利用其他节点（计算节点或数据管理节点）将这些本地内存表刷新到存储设备的效率较低，从而导致性能下降。未来，我们将研究如何自动确定本地内存表与解耦内存之间的比例。

网络 I/O 分解。 如表 5 所示，我们分析了 O^3-LSM 如何在不同节点之间重新分配网络流量。在*Disagg-RocksDB*中，计算节点（拥有 LSM 键值存储）负责将数据刷新到存储设备的所有 I/O。相比之下， O^3-LSM 可以将大量流量卸载到其他可用的计算节点或数据管理节点。在双计算节点环境下进行的 5000 万次随机写入工作负载测试中，虽然内存表卸载（计算节点→解耦内存）产生了 12.6 吉字节的流量，但我们成功将计算节点→存储设备之间的直接压力从 59.3 吉字节降至 46.1 吉字节。这些流量在两个计算节点和解耦内存节点之间得到了有效的重新平衡：两个计算节点分别承担 25.36 吉字节和 20.74 吉字节，解耦内存节点则处理剩余的 4.3 吉字节。这些结果验证了我们提出的调度算法在组件之间重新分配网络 I/O 的有效性。在写入密集型突发期间，这种转变至关重要，因为它能将网络 I/O 分散到所有可用资源，避免单个节点的网络接口成为瓶颈。

分片级优化。 表 6 展示了分片级分区在平滑网络流量方面的关键作用。没有分片级优化时，系统会出现大幅流量尖峰，峰值计算节点→解耦内存吞吐量达到 963.2 MB/s，而平均吞吐量仅为 39.2 MB/s。这种突发行为会导致严重的网络拥塞。通过将内存表划分为分片，并并行调度其传输和刷新， O^3-LSM 将峰值计算节点→解耦内存流量降低了 6.5×（从 963.2 降至 146.2 MB/s），同时提高了

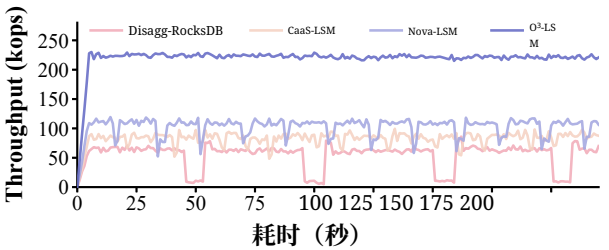


图13：随时间变化的吞吐量稳定性。

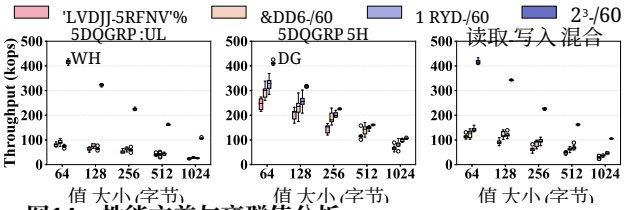


图14：性能方差与离群值分析。

工作负载	吞吐量（千操作每秒/秒）	P99延迟（μ秒）
随机写入	265.8 (60.1)	82.4 (169.1)
随机读取	255.4 (189.1)	210.1 (440.8)
读写混合	272.5 (84.8)	215.8 (528.0)

表7：启用和（未启用） O^3-LSM 优化的 CaaS-LSM。

平均吞吐量。这些结果说明，分片刷新能够缓解带宽突发，并平滑各节点之间的网络流量。因此，分片将 W100-Uniform 吞吐量从 268.1 提升至 428.9 千操作每秒（1.6×）。即使在 W100-Zipfian 齐夫偏斜下， O^3-LSM 仍可达到 409.5 千操作每秒，相较于非分片基线（241.2 千操作每秒）提升了 1.7×。这表明，并行分片刷新能够缓解热点，并通过允许独立的刷新任务并发执行来保持高性能。

性能稳定性。 如时间序列分析（图13）所示，与其他基线系统相比， O^3-LSM 表现出很高的稳定性。Disagg-RocksDB 和 Nova-LSM 的吞吐量经常骤降，且方差较大。相比之下， O^3-LSM 始终保持高而平稳的吞吐量。图14中的箱线图进一步证实了这一稳健性：在所有工作负载和值大小下， O^3-LSM 的分布最为集中，离群值也最少。这种稳定性源于以细粒度分片级卸载取代粗粒度刷新。通过将数据划分为较小的分片并调度并行传输，O³-LSM 避免了传统架构中常见的排队延迟和写入停顿。这种抵御延迟波动的能力使 O³-LSM 成为延迟敏感型云应用的理想选择。

LSM 优化的可移植性 O^3。为评估我们设计的通用性，我们将 O^3-LSM 内存表和刷新卸载机制集成到 CaaS-LSM（一种具有代表性的压实卸载 LSM键值存储）中 [51]。如表7所示，集成后的系统随机写入吞吐量达到 265.8 千操作每秒（较 CaaS-LSM 的 60.1 千操作每秒提升 4.4×），P99 延迟为 82.4 μs（较 169.1 μs 降低 51%）。这表明，内存表和刷新卸载能够带来可移植的收益。 O^3-LSM 可有效缓解压实卸载框架中的写入停顿和 I/O 争用。

工作负载	Disagg-RocksDB	CaaS-LSM	O³-LSM
仅SET	86.3 / 391	105.9 / 287	**381.2 / 177**
仅GET	125.9 / 823	159.4 / 751	**272.9 / 476**
混合10/90（Get/Set）	92.5 / 450	115.2 / 320	**355.4 / 195**
混合50/50（共置）	102.1 / 610	132.8 / 510	**315.6 / 310**
混合 50/50（远程）	72.4 / 1250	98.6 / 980	**285.4 / 480**
混合 90/10（获取/设置）	118.4 / 780	148.6 / 680	**285.2 /410**
流水线SET（16 个线程）	185.2 / 1250	235.4 / 980	**720.5 /420**

表8：KVrocks 性能对比。指标为吞吐量（千操作每秒）/ P99 延迟（毫秒）。

这些结果凸显了 *O*³-LSM 作为可移植组件的潜力：通过三层卸载，最大限度地提升分离式 LSM 树的效率。

5.4 真实应用场景分析

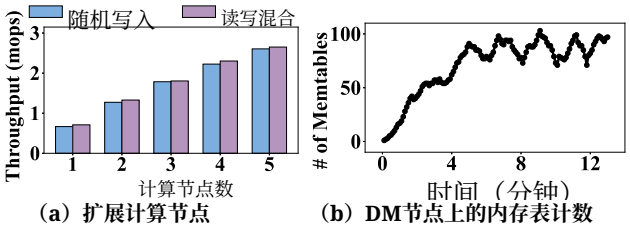
我们使用 Kvrocks [4]（一种兼容 Redis、以 RocksDB 为存储引擎的分布式键值存储）评估端到端性能。Kvrocks 集成了三种不同的解耦式 LSM 键值存储解决方案作为其存储引擎：Disagg-RocksDB、CaaS-LSM 和 O³-LSM。我们使用 Redis基准测试，测量仅GET、仅SET，以及比例为 10/90、50/50 和 90/10 的混合GET/SET工作负载的吞吐量和 P99延迟，并测试流水线版本；此外，我们还分别运行共置客户端和远程客户端。该基准测试由 16 个线程执行 4000 万次查询。如表 8 所示，在所有 Redis 基准测试工作负载中，O³-LSM 的表现始终优于 Disagg-RocksDB 和 CaaS-LSM。在仅SET和混合10/90场景中，其吞吐量最高可提升 4.4×，P99延迟降低 54%。即使在远程客户端配置和流水线工作负载下，O³-LSM 的表现也分别领先 CaaS-LSM 2.9× 和 3.0×。这些提升源于卸载内存表并并行执行分片级刷新，从而避免写入停滞和 I/O不均衡。这些结果证明了 O³-LSM 在真实应用中的高效性。

5.5 可扩展性

多计算节点的可扩展性。 我们逐步增加计算节点，在随机写入和读写混合工作负载下评估*O*³-LSM 的可扩展性。我们在不同时间间隔启动基准测试，并测量所有活动计算节点的总吞吐量。图 15a 展示了随着计算节点数量增加的吞吐量对比。对于这两种工作负载，我们都观察到吞吐量持续增长，表明其具有良好的可扩展性。再增加 4 个计算节点，最高可使总吞吐量提升 2.9

X 与单个计算节点相比。如图15b所示，在5个计算节点共享解耦内存的情况下，经过较长时间后，总内存表计数仅出现小幅波动。这种稳定性体现了更高的内存利用率和效率，与现代分布式数据中心的目标一致。通过在各计算节点之间平衡刷新操作，*O*³-LSM避免了任何单个节点同时承受I/O压力，从而确保刷新稳定。

多 DM 节点下的可扩展性。 我们评估了具有多个 DM 节点、为计算节点集群（3 个计算节点）提供服务的 *O*³-LSM。如表9所示，性能增益会随着 DM 节点数量的增加而提升。将 DM 数量从1个升级到3个后，随机写入和读写混合工作负载的总吞吐量分别提升了10.8%和11.9%。每个计算节点独立选择一个目标 DM 进行卸载，并将位置（节点ID和内存地址）存储在其本地内存表



（a）扩展计算节点　（b）DM节点上的内存表计数

图15：O³-LSM的可扩展性和内存弹性。

配置	随机写入（千操作每秒）	读写混合（千操作每秒）
3 个计算节点 +1 解耦内存	1,786	1,805
3 个计算节点 +2 解耦内存	1,920	1,950
3 个计算节点 +3 解耦内存	1,980	2,020

表9：不同数量 DM 节点下的性能。

元数据中。通过在本地维护这些跟踪信息，计算节点可以跨多个 DM 节点并发执行任务（例如内存表卸载和读取委派）。这种设计实现了卸载开销的并行化，并避免了网络 I/O 或 CPU 密集型任务中的单节点瓶颈。这种架构灵活性使 *O*³-LSM 能够线性扩展其内存层资源，以支持解耦环境中不断增加的并发请求和更大的工作集。

通过共享 DM 池缓解写入停顿。 在固定分区配置中，5 个计算节点各自在 DM 中分配到 0.5 GB 的内存配额（总计 2.5 GB）。在此设置下，单个计算节点在写入突发期间经常耗尽配额，导致总计发生 840 次停顿，P99 延迟高达 240 毫秒。相反，共享 DM 池配置利用总计 2.5 GB（相同的内存预算）容量，弹性地吸收 5 个计算节点之间不均衡的写入流量。正如表10所示，这种集体缓冲使停顿总数减少了 92.2%（从 840 次降至 65 次），P99 延迟也降低了 5× 以上。通过将内存资源与特定计算实例解耦，并将其组织为共享内存池，*O*³-LSM 允许写入强度较高的实例从池中借用闲置容量。这种弹性与并行化的分片刷新相结合，使总吞吐量保持在较高水平，达到 2.62 百万次操作/秒（每个计算节点 524 千操作每秒），较固定配置下的 1.85 百万次操作/秒（每个计算节点 370 千操作每秒）提升了 41.6%。

6 相关工作

持久化解耦式 LSM-KVS。 目前，已有许多研究致力于优化解耦存储上的持久化 LSM-KVS，包括 Kemme 等人 [1], Hailstorm [6],Nova-LSM [25], IS-HBase [9], RocksDB-Cloud [39], TerarkDB [7], 以及 Disagg-reRocksDB [17]。这些研究主要采用压实卸载或远程压实（首次提出于 [1]）来减少存储资源与计算节点之间的网络 I/O。与现有针对存储设备上的 LSM 键值存储进行优化的研究相比，O³-LSM 利用解耦内存实现内存扩展和明确的性能提升。所提出的四项主要技术也可以应用于现有技术（例如远程压实）。

面向解耦内存的内存键值存储。 已有多种内存键值存储设计针对解耦内存进行了优化。Sherman [45] 是一种基于 B+ 树的键值存储，将所有树节点置于解耦内存中，同时在计算节点本地维护元数据和索引缓存，

设置（5 个计算节点）	吞吐量（百万次操作）	写入停顿	P99延迟（毫秒）
固定池（0.5吉字节×5）	1.85	840	240
共享池（2.5吉字节）	**2.62**	**65**	**45**

表10：写入停顿缓解：固定与共享 DM 池。

并通过精心设计的远程直接内存访问模式实现高性能查找。PolarDB Serverless [8] 使用基于解耦内存的多租户内存池来缓存数据页，而 Clover [43], 、FUSEE [40], 和 Dinomo [28] 等基于哈希的系统则将所有键值对和索引结构完全存储在解耦内存中。近期的索引结构也探索了选择性远程直接内存访问卸载，以平衡单边与双边远程直接内存访问操作。例如，分布式树形索引 [54]利用这种方式加速遍历，DEX [33] 则采用类似方法，在解耦内存上进行范围索引。 O^3-LSM 独特地将这一机制融入 LSM 树的生命周期中。与这些以索引为中心的工作不同，我们的读取委派与基于预写日志的持久性和分片级刷新语义协同设计，以确保三层卸载架构中的一致性。

LSM-KVS 写入瓶颈。 多项相关研究旨在解决 LSM键值存储的写入性能瓶颈。一些研究 [18, 27] 在 LSM键值存储 的存储端增加非易失性内存层，通过支持更快的刷新来避免序列化开销并缓解写放大。与之不同，O^3-LSM 提出了新颖的远程刷新和缓存增强型读取委派，成功解决了由缓慢的解耦内存造成的性能瓶颈。多项工作 [25, 48] 对并行化 L_0-L1 压实采用了不同的设计。MatrixKV [48] 在非易失性内存上维护一个经过特殊分区的 L_0 层，旨在充分利用非易失性内存的高吞吐量和字节可寻址特性。O^3-LSM 主动合并集群中单个节点的键值范围。我们旨在将内存表卸载到解耦内存上，并充分利用与共享解耦内存相连的计算集群的充足计算资源。其他优化方案提出将压实卸载到其他计算资源，例如解耦存储节点 CPU [1, 9, 17, 25, 30, 39], 现场可编程门阵列 [24] 或其他计算单元 [15, 47]，以缓解压实开销，或通过适当的压实策略与时机 [5, 14, 50]来维持稳定性。

7 结论

本文提出了 O^3-LSM，这是首个利用解耦内存作为解耦存储写缓冲区扩展的 LSM键值存储。我们通过针对 DM 优化的内存表应对解耦内存的性能挑战，同时引入缓存增强型读取委派和分片级刷新卸载。未来的工作包括细粒度 DM 内存表管理、自动化内存比例以及高级刷新调度。

参考文献

[1] Muhammad Yousuf Ahmad 和 Bettina Kemme。2015年。《分布式键值数据存储中的压缩管理》。（2015年），850–861。doi:10.14778/2757807.275 7810[2] Emmanuel Amaro、Christopher Branner-Augmon、Zhihong Luo、Amy Ouster- hout、Marcos K. Aguilera、Aurojit Panda、Sylvia Ratnasamy 和 Scott Shenker。2020年。远程内存能否提高作业吞吐量？载于第*15*届欧洲计算机系统会议论文集，*EuroSys 2020*（2020-04-15）（第*15*届欧洲计算机系统会议论文集，*EuroSys 2020*）。计算机协会。doi: 10 1145 3342195 3387522 。/。[3] Sebastian Angel、Mihir Nanavati 和 Siddhartha Sen。。解聚与应用。载于第*12*届 *USENIX* 云计算热点会议论文集（美国，2020年）（*HotCloud'20*）。USENIX 协会，文章15。2020

[4] Apache . [n. d.]. Kvrocks . https://github .com/apache/incubator - kvrocks[5] Oana Balmau、Florin Dinu、Willy Zwaenepoel、Karan Gupta、Ravishankar Chand- hiramoorthi 和 Diego Didona。2019。SILK：防止日志结构合并键值存储中的延迟尖峰。载于*2019*年*USENIX*年度技术会议（*USENIX ATC 19*）（美国华盛顿州伦顿，2019-07）。USENIX 协会，753–766。https://www.usenix.org/conference/atc19/presentation/balmau [6] Laurent Bindschaedler、Ashvin Goel 和 Willy Zwaenepoel。2020。Hailstorm：面向分布式LSM数据库的解聚计算与存储。载于第二十五届编程语言与操作系统架构支持国际会议论文集（瑞士洛桑，2020）（*Asplos '20*）。计算机协会，纽约，美国，301–316。doi:10.1145/3373376.3378504[7] 字节跳动。2019。*Terark*。https://github.com/bytedance/terarkdb[8] Wei Cao、Yingqiang Zhang、Xinjun Yang、Feifei Li、Sheng Wang、Qingda Hu、Xuntao Cheng、Zongzhi Chen、Zhenjun Liu、Jing Fang、Bo Wang、Yuhui Wang、Haiqing Sun、Ze Yang、Zhushi Cheng、Sen Chen、Jian Wu、Wei Hu、Jianwei Zhao、Yusong Gao、Songlu Cai、Yunyang Zhang 和 Jiawang Tong。2021。PolarDB Serverless：面向解耦数据中心的云原生数据库。载于*2021*年数据管理国际会议论文集（线上活动，中国，2021）（*Sigmod '21*）。计算机协会，纽约，美国，2477 2489 10 1145 3448016 3457560 –。doi:。/。[9] Zhichao Cao、Huibing Dong、Yixun Wei、Shiyong Liu 和 David H. C. Du。2022。IS-hbase：针对计算存储解聚基础设施进行优化、具备 I/O 卸载和自适应缓存功能的存储内计算 Hbase。18, 2，文章 15 2022 10 1145 3488368 （）。doi:。/。[10] Zhichao Cao、Siying Dong、Sagar Vemuri 和 David HC Du。2020。对脸书 RocksDB 键值工作负载进行特征分析、建模与基准测试。载于第十八届*USENIX*文件与存储技术会议（*FAST 20*）（2020）。209–223。[11] Compute Express Link（CXL）联盟。2019。*CXL*联盟。https: //www.computeexpresslink.org [12] Compute Express Link（CXL）联盟。访问日期：2022年11月。CXL 3.0白皮书。https://www.computeexpresslink.org/ _ 0 1418_ 8713008 files/ugd/ c a ae d a b.pdf[13] Brian F Cooper、Adam Silberstein、Erwin Tam、Raghu Ramakrishnan 和 Russell Sears。2010。使用 YCSB 对云服务系统进行基准测试。载于第一届 *ACM*云计算研讨会论文集（2010）。ACM，143–154。[14] Niv Dayan、Tamar Weiss、Shmuel Dashevsky、Michael Pan、Edward Bortnikov 和 Moshe Twitto。2022。Spooky：正确地细分LSM树压实。15，11（2022），3071–3084。doi:10.14778/3551793.3551853[15] Chen Ding、Jian Zhou、Jiguang Wan、Yiqin Xiong、Sicen Li、Shuning Chen、Hanyang Liu、Liu Tang、Ling Zhan、Kai Lu 和 Peng Xu。2023。DComp：使用数据处理单元高效卸载LSM树压实。载于第五十二届并行处理国际会议论文集（¡conf-loc¿，¡city¿盐湖城¡/city¿，¡state¿UT¡/state¿，¡country¿美国¡/country¿，¡/conf-loc¿，2023）（*Icpp '23*）。计算机协会，纽约，美国，233–243。doi:10.1145/3605573.3605633[16] Siying Dong、Mark Callaghan、Leonidas Galanis、Dhruba Borthakur、Tony Savor 和 Michael Strum。2017。优化 RocksDB 中的空间放大。载于*CIDR*（2017），第3卷。3。[17] Siying Dong、Shiva Shankar P、Satadru Pan、Anand Ananthabhotla、Dhanabal Ekambaram、Abhinav Sharma、Shobhit Dayal、Nishant Vinaybhai Parikh、Yanqin Jin、Albert Kim、Sushil Patil、Jay Zhuang、Sam Dunster、Akanksha Mahajan、Anirudh Chelluri、Chaitanya Datye、Lucas Vasconcelos Santana、Nitin Garg 和 Omkar Gawde。2023。RocksDB解聚：生产实践。1，2，文章 192 2023 10 1145 3589772 （）。doi:。/ [18] Zhuohui Duan、Jiabo Yao、Haikun Liu、Xiaofei Liao、Hai Jin 和 Yu Zhang。2023。重新审视混合内存系统中的键值存储日志结构合并。载于第二十八届*ACM*编程语言与操作系统架构支持国际会议论文集，第2卷（加拿大不列颠哥伦比亚省温哥华，2023）（*Asplos 2023*）。计算机协会，纽约，美国，674–687。doi:10.1145/3575693.3575715[19] Dmitry Duplyakin、Robert Ricci、Aleksander Maricq、Gary Wong、Jonathon Duerig、Eric Eide、Leigh Stoller、Mike Hibler、David Johnson、Kirk Webb、Aditya Akella、Kuangching Wang、Glenn Ricart、Larry Landweber、Chip Elliott、Michael Zink、Emmanuel Cecchet、Snigdhaswin Kar 和 Prabodh Mishra。2019。CloudLab 的设计与运行。载于*2019*年*USENIX*年度技术会议（*USENIX ATC 19*）（美国华盛顿州伦顿，2019-07）。USENIX 协会，1–14。https: // www.usenix.org/conference/atc19/presentation/duplyakin [20] 脸书。2021。*ZippyDB*。https://engineering.fb.com/2021/08/06/core- infra/zippydb/ [21] Peter X. Gao、Akshay Narayan、Sagar Karandikar、Joao Carreira、Sangjin Han、Rachit Agarwal、Sylvia Ratnasamy 和 Scott Shenker。2016。资源解聚的网络需求。载于第十二届*USENIX*操作系统设计与实现研讨会（*OSDI 16*）（美国佐治亚州萨凡纳，2016-11）。USENIX 协会，249–264。https://www.usenix.org/conference/osdi16/technical- sessions/presentation/gao [22] Sanjay Ghemawat 和 Jeff Dean。2011。LevelDB。（2011）。

[23] Sanjay Ghemawat、Howard Gobioff 和 Shun-Tak Leung。2003。谷歌文件系统。载于第十九届ACM操作系统原理研讨会论文集（2003）。29–43。[24] Gui Huang、Xuntao Cheng、Jianying Wang、Yujie Wang、Dengcheng He、Tieying Zhang、Feifei Li、Sheng Wang、Wei Cao、Qiang Li。2019。X-Engine：面向大规模电子商务事务处理的优化存储引擎。载于2019年数据管理国际会议论文集（荷兰阿姆斯特丹，2019）（Sigmod '19）。计算机协会，纽约，美国，651 665 10 1145 3299869 3314041 –。doi:。/。[25] Haoyu Huang 和 Shahram Ghandeharizadeh。2021。。Nova-LSM：一种分布式、组件化的LSM树键值存储。载于2021年数据管理国际会议论文集（2021-06）。ACM。doi:10.1145/3448016. 3457297[26] IBM。2018。解耦内存。https://www.ibm.com/blogs/research/2018/01/advancing-cloud-memory-disaggregation/[27] Wonbae Kim、Chanyeol Park、Dongui Kim、Hyeongjun Park、Young-ri Choi、Alan Sussman 和 Beomseok Nam。2022。ListDB：预写日志与持久化跳表的结合，用于持久内存上的增量检查点。载于第十六届USENIX操作系统设计与实现研讨会（OSDI 22）（美国加利福尼亚州卡尔斯巴德，2022-07）。USENIX 协会，161–177。https://www.usenix.org/conference/osdi22/presentation/kim[28] Sekwon Lee、Soujanya Ponnapalli、Sharad Singhal、Marcos K. Aguilera、Kimberly Keeton 和 Vijay Chidambaram。2022。DINOMO：面向解耦持久内存的弹性、可扩展、高性能键值存储。15，13（2022），4023–4037。doi:10.14778/3565838.3565854[29] Huaicheng Li、Daniel Berger、Stanko Novakovic、Lisa Hsu、Dan Ernst、Pantea Zardoshti、Monish Shah、Ishwar Agarwal、Mark Hill、Marcus Fontoura 和 Ricardo Bianchini。2022。面向云平台的第一代内存解耦。（2022）。[30] Jianchuan Li、Peiquan Jin、Yuanjin Lin、Ming Zhao、Yi Wang 和 Kuankuan Guo。2021。面向LSM树的弹性稳定压实：一种基于函数即服务的方法，应用于TerarkDB。载于第三十届ACM 信息与知识管理国际会议论文集（线上活动，澳大利亚昆士兰，2021）（Cikm '21）。计算机协会，纽约，美国，3906–3915。doi:10.1145/3459637.3481913[31] Rui Lin、Yuxin Cheng、Marilet De Andrade、Lena Wosinska 和 Jiajia Chen。2020。解耦数据中心：挑战与权衡。58，2（2020），20–26。doi:10.1109/MCOM.001.1900612[32] Xinxin Liu、Yu Hua 和 Rong Bai。2021。适用于远程持久内存且兼容 RDMA 的一致性哈希。载于2021年IEEE第39届国际计算机设计会议（ICCD）（2021）。IEEE，174–177。doi:10.1109/ICCD53106.2021.000 37[33] Baotong Lu、Kaisong Huang、Chieh-Jan Mike Liang、Tianzheng Wang 和 Eric Lo。2024。DEX：解耦内存上的可扩展范围索引。VLDB 基金会会议录17，10（2024），2603–2616。[34] Patrick MacArthur 和 Robert D. Russell。2012。指导 RDMA 编程决策的性能研究。载于2012年IEEE第14届高性能计算与通信国际会议暨2012年IEEE第9届嵌入式软件与系统国际会议论文集（美国，2012）（Hpcc ' *12* 778 785 10 1109 2012 110 ）。IEEE计算机学会，–。[35] doi:。/HPCC.。Santhosh Nagaraj Nag。2023。。CXL（Compute Express Link）技术。11，6（2023），94–102。[36] Oracle。2012。InfiniBand。https://www.oracle.com/technetwork/server-storage/networking/documentation/o12-020-1653901.pdf[37] Satadru Pan、Theano Stavrinos、Yunqiao Zhang、Atul Sikaria、Pavel Zakharov、Abhinav Sharma、Shiva Shankar P、Mike Shuey、Richard Wareing、Monika Gangapuram、Guanglei Cao、Christian Preseau、Pratap Singh、Kestutis Patiejunas、given-i=JR family=Tipton、given=JR、Ethan Katz-Bassett 和 Wyatt Lloyd。2021。脸书的 Tectonic 文件系统：百亿亿级规模下的效率。载于第十九届USENIX文件与存储技术会议（FAST 21）（2021-02）。USENIX 协会，217–231。https://www.usenix.org/conference/fast21/presentation/pan[38] Gangqi Huang Qi Lin。2024。https://github.com/argetterxx/O3-LSM。[39] rockset。2020。Rockscloud。https://github.com/rockset/rocksdb-cloud[40] Jiacheng Shen、Pengfei Zuo、Xuchuan Luo、Tianyi Yang、Yuxin Su、Yangfan Zhou 和 Michael R. Lyu。2023。FUSEE：一种完全采用内存解耦的键值存储。载于第二十一届USENIX文件与存储技术会议论文集（美国加利福尼亚州圣克拉拉，2023）（FAST'23）。USENIX 协会，美国，文章6。[41] Vishal Shrivastav、Asaf Valadarsky、Hitesh Ballani、Paolo Costa、Ki Suh Lee、Han Wang、Rachit Agarwal 和 Hakim Weatherspoon。2019。Shoal：面向解耦式机架的网络架构。载于第十六届USENIX网络系统设计与实现研讨会（NSDI 19）（美国马萨诸塞州波士顿，2019-02）。USENIX 协会，255–270。https://www.usenix.org/conference/nsdi19/presentation/ shrivastav[42] Konstantin Shvachko、Hairong Kuang、Sanjay Radia 和 Robert Chansler。2010。Hadoop 分布式文件系统。载于2010年IEEE第26届大容量存储系统与技术研讨会（MSST 10）（2010）。IEEE，1–10。

[43] Shin-Yeh Tsai、Yizhou Shan 和 Yiying Zhang。2020年。《解耦持久内存并进行远程控制：对被动解耦键值存储的探索》。载于《*2020*年 *USENIX* 年度技术会议论文集》（美国，2020年）（*Usenix Atc'20*）。USENIX 协会，文章 3。[44] Jianguo Wang 和 Qizhen Zhang。2023年。《解耦数据库系统》。载于《*2023*年数据管理国际会议伴随论文集》（2023年）。37–44。[45] Qing Wang、Youyou Lu 和 Jiwu Shu。2022年。《Sherman：一种面向解耦内存、针对写入优化的分布式 B+树索引》。载于《*2022*年数据管理国际会议论文集》（美国宾夕法尼亚州费城，2022年）（*Sigmod '22*）。计算机协会，纽约，纽约州，美国，1033–1048。doi:10.1145/3514221.3517824[46] Ruihong Wang、Jianguo Wang、Prishita Kadam、M Tamer Özsu 和 Walid G Aref。2023年。《dLSM：一种用于内存解耦的基于 LSM 的索引》。载于《*2023*年 *IEEE* 第*39*届国际数据工程会议（*ICDE*）》（2023年）。IEEE，2835–2849。[47] Peng Xu、Jiguang Wan、Ping Huang、Xiaogang Yang、Chenlei Tang、Fei Wu 和 Changsheng Xie。2020 年。《 LUDA：借助 GPU 加速 LSM 键值存储压实》。arXiv: 2004 03054 [ ] 。cs.DC Ting Yao、Yiwen Zhang、Jiguang Wan[48]、Qiu Cui、Liu Tang、Hong Jiang、Changsheng Xie 和 Xubin He。2020年。《MatrixKV：利用非易失性内存中的矩阵容器，减少基于 LSM 树的键值存储中的写入停顿和写放大》。载于《*2020*年 *USENIX* 年度技术会议论文集》（美国，2020年）（*Usenix Atc'20*）。USENIX 协会，文章 2。[49] Seehwan Yoo、Hojin Shin、Sunghyun Lee 和 Jongmoo Choi。2022年。《现代键值存储中存储层次结构的读取性能分析：以 RocksDB 为例》。载于《*2022*年 *IEEE* 第*11*届非易失性内存系统与应用研讨会（*NVMSA*）》（2022 45 50 10 1109 56066 2022 00017 ）。–。doi:./NVMSA。[50] Jinghuan Yu、Sam H. Noh、Young-ri Choi 和 Chun Jason Xue。2023年。《ADOC：自动协调日志结构键值存储各组件间的数据流以提升性能》。载于《第*21*届 *USENIX* 文件与存储技术会议（*FAST 23*）》（加利福尼亚州圣克拉拉，2023年2月）。USENIX 协会，65–80。https://www.usenix.org/conference/fast23/presentation/yu[51] Qiaolin Yu、Chang Guo、Jay Zhuang、Viraj Thakkar、Jianguo Wang 和 Zhichao Cao。2024年。《CaaS-LSM：面向存储解耦基础设施中基于 LSM 的键值存储的压实即服务》。2，3（2024年），1–26。[52] Qizhen Zhang、Yifan Cai、Sebastian Angel、Ang Chen、Vincent Liu 和 Boon Thau Loo。2020年。《重新思考解耦数据中心的数据管理系统》。载于《创新数据系统研究会议》（2020年）。[53] Qizhen Zhang、Yifan Cai、Xinyi Chen、Sebastian Angel、Ang Chen、Vincent Liu 和 Boon Thau Loo。2020年。《理解数据中心资源解耦对生产环境数据库管理系统的影响》。13，9 3 397230 2020 1568 1581 10 14778 3397249（），–。doi:./。[54] Tobias Ziegler、Sumukha Tumkur Vani、Carsten Binnig、Rodrigo Fonseca 和 Tim Kraska。2019年。《为支持 RDMA 的高速网络设计分布式树型索引结构》。载于《*2019*年数据管理国际会议论文集》。741–758。