Nezha：优化 Raft 集成的键值分离分布式存储

Yangyang Wang[1,2], Yucong Dong[1,2],Ziqian Cheng[1,2],和 Zichen Xu[1,2*]

[1]南昌大学人工智能学院，中国[2]南昌大学数学与计算机科学学院，中国

[*]通讯作者：xuz@ncu.edu.cn

arXiv:2603.09122v1 [cs.DC] 10 Mar 2026

摘要—**分布式键值存储被广泛用于支撑弹性大数据应用，并借助 Raft 等专用共识算法来确保数据一致性。然而，通过系统分析，我们发现此类一致性存储存在一个关键性能问题，即共识协议与底层存储引擎之间重叠的持久化操作会造成显著的 I/O 开销。为解决该问题，我们提出 *Nezha*，一种原型分布式存储系统，创新性地将键值分离与 Raft 相结合，从而在强一致性保证下提供可扩展的吞吐量。*Nezha*在操作层面重新设计了持久化策略，并引入分层垃圾回收，在保持 Raft 安全性属性的同时显著提升读写性能。实验结果表明，平均而言，*Nezha* 在写入、读取和 scan 操作上的吞吐量分别提升了 460.2%、12.5% 和 72.6%。**

索引词——**共识、键值存储、Raft、键值分离**

I. 引言

随着大数据应用的爆炸式增长，分布式键值存储系统已成为现代数据密集型应用不可或缺的基础设施 [1]–[4]。键值存储提供了一种简单而强大的数据管理方式，直接支持多个领域的大规模应用，包括电子商务 [5],社交网络 [6], 和机器学习 [7]。在多种实现方案中，基于日志结构合并树（LSM 树） [8] 的系统因能够通过顺序磁盘I/O优化写操作而得到广泛应用 [2]–[4], [9], [10]。然而，这种写入优化设计不可避免地要以牺牲读取性能为代价：LSM 树的多层结构使得即使是简单的点查询，也需要遍历多个文件 [11]。与此同时，压实过程引入的写放大显著降低了存储效率 [11], [12]。

分布式存储系统面临着同时维持高性能和数据一致性的双重挑战。后一个挑战通常通过分布式共识协议来解决，尤其是 Paxos [13], [14] 和 Raft[15], [16],，它们确保多个节点之间的数据副本保持一致。由于设计易于理解且实现直接，Raft 已成为生产系统中的首选协议，包括 etcd [17], 、CockroachDB [18],、TiDB [19], 和 PolarFS [20], 。然而，Raft 的安全性保证要求对客户端请求进行持久化存储，因此在分布式键值存储系统中实现时会产生冗余的磁盘写入。

研究发现概要。 在采用 LSM 树 [8] 存储引擎的基于 Raft 的分布式键值存储中，每次数据写入操作至少需要进行三次不同的磁盘写入：一次用于 Raft 日志持久化，一次用于 LSM 树的预写日志（WAL），另一次用于内存表持久化。此外，LSM 树的后台压缩过程会带来额外的磁盘写入开销，它会持续合并并重写多个层级中的数据。尽管这些冗余写操作能够确保数据持久性，但我们的分析表明，并非所有数据都始终需要如此全面的持久化，这意味着存在性能优化的潜在机会。

工作概述。为减少这些冗余写操作，同时保留数据一致性保证，我们提出了*Nezha*，一种新型分布式键值存储系统，通过架构创新实现全面的性能提升。我们的工作包含以下创新设计：1）针对写请求，我们提出 KVS-Raft，这是一种专为分布式键值存储优化的共识算法，通过将键值分离策略与 *Raft* 协议有机结合，在保持 Raft 安全性属性的同时减少写放大；2）针对读请求，我们设计了感知 *Raft* 的垃圾回收（*GC*）框架，以平衡读取性能；3）为确保请求处理的正确性，我们实现了三阶段请求处理机制，确保系统在不同优化阶段正确处理读写请求。

为验证我们的设计，我们实现了 Nezha，并进行了全面的实验评估。实验结果表明，与传统的基于 Raft 的键值存储系统相比，性能显著提升：写入、读取和扫描操作的平均吞吐量分别提高了 *460.2%*、*12.5%* 和 *72.6%*。

本文的主要贡献包括：

- 我们发现，分布式键值存储系统中的传统共识协议实现会引入不必要的数据持久化操作，这些操作在现代硬件环境中已成为性能瓶颈。

- 我们设计了 Nezha，一种新型分布式键值存储系统。在共识层，我们提出了 KVS-Raft，这是一种增强型共识协议，在保持强一致性的同时集成了键值分离。在存储层，我们设计了一个 Raft 感知型垃圾回收框架，并辅以三阶段请求处理机制，以确保系统优化期间正确处理请求。
- 我们实现了 Nezha，并进行了广泛的实验评估，结果表明，在各种工作负载模式下，其性能相较传统基于 Raft 的存储系统均有显著提升。

本文其余部分组织如下。第二节介绍背景与动机。第三节阐述 Nezha 的设计。第四节描述系统实现与评估。第五节讨论相关工作。最后，第六节对全文进行总结。

II. 背景与动机

本节为我们的研究奠定基础。我们首先在第二节-A 中介绍分布式共识与数据持久化的基本概念，随后在第二节-B 中全面回顾 Raft 协议。第二节-C 介绍键值分离，这一基础技术启发了我们的设计。在第二节-D 中，我们指出当前分布式键值存储系统中的关键性能瓶颈，尤其是冗余数据持久化问题，并探讨将键值分离与 Raft 相结合的潜在收益。

A. 共识算法与数据持久化

定义 2.1。（分布式共识）分布式共识是分布式系统中的一个基本问题，要求系统中的多个节点就某个值或状态达成一致，即使存在节点故障、网络分区及其他故障场景也是如此。

定义 2.2。（数据持久化）数据持久化是指将数据从易失性存储（例如内存）转移到非易失性存储设备（例如固态硬盘、机械硬盘）的过程。此过程依赖文件系统写操作和同步机制，以确保数据持久性和一致性语义。

分布式共识是构建可靠分布式系统的基石。在各种分布式系统应用中，确保所有节点就关键数据和操作达成一致至关重要 [21]–[23]。例如，分布式存储系统必须确保节点就
数据修改操作的全序 [24]。此一致性要求直接影响系统正确性和可用性。

为维持此类一致性保证，数据持久化是确保可靠性的关键机制。尽管现代存储系统通常采用内存缓存来提升性能 [25]–[27], ，但内存的易失性要求定期执行持久化操作，以防止数据丢失，这是由于

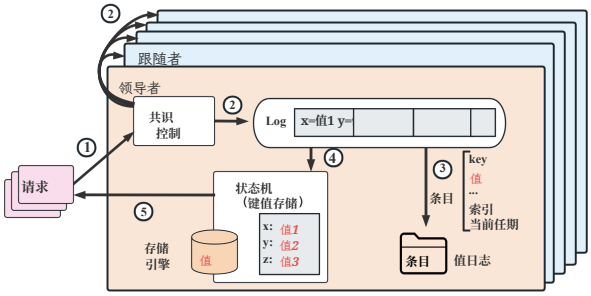


图 1. Raft 请求处理示例。电源故障或系统崩溃。该过程通常涉及 fsync() 等系统调用，会带来显著的 I/O 开销。因此，在持久化操作的频率与性能开销之间取得平衡，成为系统设计中的一项关键考量。

在实际系统实现中，分布式共识与数据持久化这两项基本需求紧密耦合。为确保故障恢复后的系统一致性，共识算法通常要求将已达成一致的数据写入持久化存储。该持久化机制不仅支持节点重启后的状态恢复，也为新节点加入系统时的数据同步奠定基础。尽管这种“共识-持久化”耦合模式已成为分布式系统设计中的标准范式，但它也带来了显著的性能挑战。

B. Raft 协议

Raft [15] 通过将分布式共识分解为三个相对独立的子问题：领导者选举、日志复制和安全性保证，显著简化了其复杂性。这种设计方法使其在许多实际系统中得到广泛采用，例如 etcd[17]、CockroachDB [18]、TiDB [19], 和 PolarFS [20]。

Raft 采用基于领导者的方法达成共识。一个节点被选为领导者，并全权负责管理复制日志。其他节点则作为跟随者，如图 1 所示。在 Raft 的标准实现中，对系统状态的任何修改都必须经过严格的协议流程：领导者节点首先接收客户端请求（步骤 ①），将请求内容作为新的 Raft 日志条目追加到其本地日志中，同时通过日志复制机制将其同步到集群中的跟随者（步骤 ②）。为确保算法的安全属性，Raft 强制执行以下关键约束：

- 日志持久化：所有日志条目都必须持久化写入值日志（用于持久化 Raft 日志条目的专用磁盘文件），以防止节点崩溃时发生数据丢失（步骤 ③）。
- 多数确认：只有在获得集群中多数节点确认后，日志条目才能提交到状态机（步骤 ④-⑤）。
- 状态恢复：节点必须能够在重启后从持久化存储中完全恢复已提交的日志。

如图 1 中“值”所示，这些安全要求导致数据持久化操作频繁发生。尤其是在

分布式键值存储系统中，这些操作与存储引擎自身的持久化机制交织在一起，从而带来额外的性能开销。

C. 键值分离

键值分离是一种存储优化技术，旨在减少基于 LSM 树的系统中的写放大。WiscKey [11] 开创了这一架构，展示了相较于传统 LSM 树设计的显著性能提升。其核心思想是只在 LSM 树中存储键和值指针，而将实际值存放在单独的仅追加日志文件中。在压实期间，只需重写轻量级的键和指针，从而避免在 LSM 树层级之间反复移动较大的值。这种设计显著降低了写放大，也缩小了 LSM 树的大小，从而提升写入性能和缓存效率。

D. 动机

共识协议与存储引擎的协同设计，近来已成为分布式存储系统中的一个重要研究方向。PASV [28] 发现了 Raft 与 LSM 树之间的双重日志记录问题，并提出移除存储层预写日志。PALF [29]重新设计了面向分布式数据库的复制式预写日志。最近，LSM-Raft [30] 通过传输压实后的有序字符串表（SSTable）文件，而非细粒度日志条目，优化了跟随者端的冗余写入。这些近期研究表明，共识层与存储层之间的冗余持久化已成为关键的性能瓶颈。

与此同时，近期的硬件演进进一步加剧了这一问题的紧迫性。高性能 NVMe 固态硬盘的普及，使软件层面的 I/O 效率低下成为新的瓶颈。此外，在采用按使用量计费的存储的云原生环境中，消除冗余写入可直接节省成本。

冗余持久化操作分析。受这些趋势的启发，我们深入分析了现有基于 Raft 的分布式键值存储系统中的冗余持久化问题。Raft 协议要求将每个请求作为日志条目持久化存储，以保留共识状态，并在节点重启后实现快速恢复与同步。与此同时，存储引擎会实施各自的持久化机制，以确保数据可靠性

并支持故障恢复。如图2所示，在基于 Raft 和 LSM 树的分布式键值存储系统中，每个写请求都必须经历以下关键持久化操作：1) 将请求内容序列化并写入持久化Raft日志；2) 达成共识后写入 LSM 树的预写日志（WAL）；3) 当 MemTable（内存表）达到其容量阈值时，将数据刷新到排序字符串表（SST、SSTable）文件中。请注意，本分析聚焦于基于 LSM 树的分布式存储系统，本文不将讨论扩展到其他存储架构。

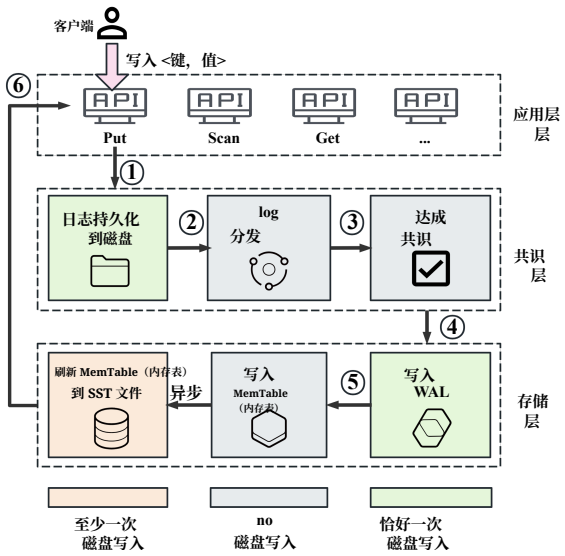


图 2. 基于 Raft 的分布式键值存储采用 LSM 树存储引擎时写操作的处理流程。

Raft 与键值分离的集成机会。为解决系统中的数据冗余持久化问题，我们发现了一个独特的优化机会：将键值分离（第二节 C 中介绍）与 Raft 协议相结合。通过在存储引擎中仅存储轻量级偏移量，同时将值保留在 Raft 日志中，我们可以将值持久化与日志复制统一起来，从而消除存储层的冗余持久化操作。

通过将键值分离与 Raft 的日志复制机制进行创新性整合，我们提出了一种新颖的存储架构。在这一设计中，值仅存储在 Raft 日志中，从而将值持久化与日志复制统一起来，消除了存储层的冗余持久化操作。然而，这种创新架构带来了两个关键技术挑战：

- 安全性保证下的垃圾回收：Raft 通过维护日志顺序性和完整性来确保数据一致性，但键值存储系统中现有的垃圾回收机制可能会损害这些基本属性。
- 读取性能优化：键值分离固有的特性使得在处理读请求（读取和扫描操作）时，必须增加额外的 I/O 开销来重建键值关联，这可能会显著降低读取性能。

为应对这些挑战，我们设计了一种 Raft 感知型垃圾回收框架，在读取性能和写性能之间实现动态平衡。该框架能够精确控制垃圾回收的触发时机，将分散的有效数据重新组织到顺序排列的磁盘文件中，并构建高效的索引结构以加速数据访问。具体而言，我们针对点查询和范围查询分别优化性能：通过维护键到偏移量映射的哈希索引来加速点查询，而

数据的顺序组织则提高了范围查询效率。该方案不仅缓解了键值分离导致的读取性能下降，还通过与 Raft 日志管理机制深度集成，确保垃圾回收过程的一致性和可靠性。最终，我们的方法在保持键值分离所带来的写效率优势的同时，有效克服了其固有的读取性能局限。

III. NEZHA

基于对分布式键值系统中共识协议与存储引擎交互的分析（第二节），我们提出了 Nezha，一种新型分布式键值存储系统。该系统通过三项关键架构创新实现了高性能：1）存储优化型共识机制，仅在值日志中持久化值和 Raft 相关元数据一次，从而显著降低写入 I/O 开销；2）系统化的垃圾回收框架，在优化存储效率的同时大幅提升读取性能；3）三阶段请求处理机制，确保在整个存储生命周期中正确、高效地处理请求。第三节 B、C 和 D 分别详细介绍了这些系统组件的设计。第三节 E 讨论了 Nezha 的容错能力和安全性保证。

A. 概述

现代分布式键值存储系统（例如 TiKV[31], ）在概念上可划分为三个基本层：应用层、共识层和存储层。如图 3 所示，Nezha 采用了这种分层架构。在应用层，系统提供全面的键值存储接口，包括写入、读取和扫描操作，不仅满足多样化的数据存储需求，还确保与现有系统兼容。在共识层，我们提出了 KVS-Raft 算法，将键值分离策略创新性地融入 Raft 共识协议，以优化系统性能。在存储层，我们设计了一种与 KVS-Raft 深度适配的存储管理机制。我们的核心贡献是设计了一套垃圾回收框架，将请求处理划分为三个不同阶段：垃圾回收前、垃圾回收期间和垃圾回收后。这种细粒度的阶段划分确保在垃圾回收期间正确处理并发用户请求，在保持强一致性的同时保证系统可用性。

通过在共识层与存储层深度集成键值分离机制，Nezha不仅在分布式环境中严格保证强一致性，还实现了显著的性能优化。这种跨层协同优化策略充分运用了键值分离的优势，同时有效缓解了其潜在局限。

B. 优化的*Raft*日志存储机制

在传统的分布式键值存储系统中，共识算法在写请求处理方面面临显著的性能瓶颈。

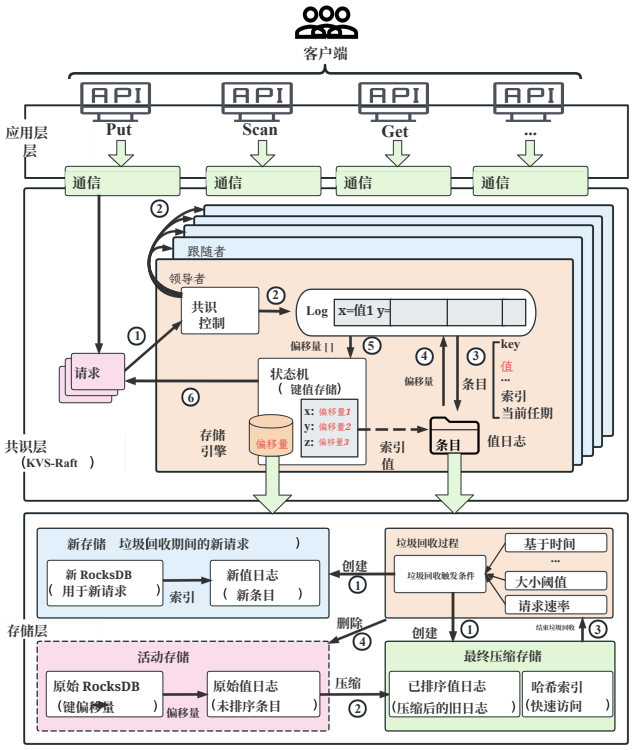


图3. Nezha的架构。

其根本原因在于，为确保一致性和持久性，大量值数据需要执行多次磁盘写操作，这导致 I/O 开销过高，并显著增加了请求处理延迟。为应对这一挑战，我们提出了一种将 Raft 协议与键值分离策略相结合的创新方法。

具体而言，如图 3 的**共识层** 所示，KVS-Raft 通过以下写请求处理流程实现了这一优化：首先，客户端写请求到达领导者节点的共识控制模块（步骤 ①）。该模块将写请求封装为 Raft 日志条目，在本地追加日志的同时将其同步给跟随者（步骤 ②）。Raft 日志追加完成后，系统将键值对及其共识相关元数据（例如当前任期和索引）序列化为一个条目 实体，将其持久化到 ValueLog 文件中（步骤 ③），随后获取该条目对应的文件偏移量（步骤 ④）。之后，领导者节点只需将轻量级偏移量（而非大体积的值）应用到其状态机中（步骤 ⑤）。最后，系统向客户端返回响应，完成写操作（步骤 ⑥）。

这种优化后的存储架构带来了多项关键优势。首先，由于值数据只需写入值日志一次，并且状态机中只维护轻量级偏移量，系统将值数据的磁盘写入次数从传统方法中的至少三次减少到仅一次，显著降低了 I/O 开销。其次，轻量级状态机设计（仅存储偏移量，而非完整

的值）不仅减少了存储空间占用，还加快了状态更新和恢复过程，正如我们的实验评估（第四节-H）所示。更重要的是，将键值分离集成到 Raft 共识层，在完全保留 Raft 安全性属性的同时提升了写性能，为构建高性能分布式键值存储系统提供了一种切实可行的方案。

C. 自适应存储管理机制

虽然针对 Raft 日志优化的存储机制显著提升了写性能，但也带来了两个关键技术挑战。首先，Raft 对严格日志排序的要求（这是实现共识协议一致性和状态复制的必要条件）与键值分离存储的非顺序特性存在固有冲突，因此传统的垃圾回收[11] 并不适用，因为它们可能会损害 Raft 日志的完整性和顺序性。其次，键值分离在处理读请求时会带来额外的 I/O 开销，因为重建键值关联需要额外的磁盘操作。为应对这些挑战，我们设计了一个自适应存储管理框架，如图 3 的**存储层**所示。该框架包含两种核心存储文件：

- 无序日志文件：这些文件以仅追加的时间顺序存储初始 Raft 日志条目，保留完整的日志信息，以支持 Raft 复制与恢复机制。
- 有序日志文件：这些文件存储经过排序和压缩的 Raft 日志条目，采用基于键的顺序存储，以优化范围查询性能，同时消除冗余和过期数据。

活动存储模块作为主要写入接收端，负责处理实时读写请求并维护无序日志文件。得益于 KVS-Raft 的优化，该模块只需在 RocksDB [2]（一种流行的单机基于 LSM 树的存储引擎）中存储轻量级偏移量，而不必存储完整的值，从而实现高效的写入支持。为有效管理累积的日志文件，系统会根据多维触发条件启动垃圾回收，包括存储空间阈值、定时机制和请求负载水平。尽管这种全面的触发机制能够确保高效的存储利用率和稳定的系统性能，但系统在垃圾回收期间必须保持读写可用性，因此需要额外的存储组件。

新存储模块专为在垃圾回收期间处理读写请求而设计。与活动存储类似，它由一个用于存储新条目的 ValueLog 文件和一个用于维护轻量级键-偏移量映射的 RocksDB 实例组成。系统启动垃圾回收后，所有新的读写请求都会转发到该模块；通过维护独立的索引结构，确保系统持续可用。这一设计使垃圾回收能够在不影响系统可用性的情况下进行，同时为后续的数据合并划定明确的边界。

表 I 不同系统阶段的存储模块组合。

阶段	存储模块
垃圾回收前	活动存储
垃圾回收期间	新存储，活动存储
垃圾回收后	新存储，最终压缩存储

最终压缩存储模块作为垃圾回收的输出，用于存储经过重组和压缩的有序数据。该模块结合哈希索引结构与排序数据固有的顺序布局，显著提升了范围查询性能。值得注意的是，已排序值日志反映了与活动存储中原始值日志相对应的状态机状态。我们的系统通过将已排序值日志与原始值日志中的两个变量——最后一个索引和最后一个任期——相结合来创建快照，这与 ***Raft*** 论文中所述的日志压缩机制一致 ***[32]***。因此，垃圾回收完成后，系统可以安全地删除活动存储中的旧 ValueLog 文件及相应的 RocksDB 文件。此时，最终压缩存储不仅通过高效的数据布局和压缩策略减少存储空间占用，还通过基于快照的恢复机制确保崩溃一致性和系统状态完整性。

如图 3 的**存储层**所示，系统在各个阶段采用不同的存储模块组合。这种系统化的模块化架构提供了清晰的数据边界，使读写操作能够精确指向合适的存储组件。为了系统地分析垃圾回收框架，我们将其生命周期分解为以下关键阶段：（1）垃圾回收初始化：系统创建新存储以处理传入的写请求，同时在最终压缩存储中初始化已排序值日志，用于存储有序数据条目（步骤①）；（2）数据压缩：系统通过压缩与排序操作对活动存储中的值日志进行重组，并在最终压缩存储中生成有序值日志。与此同时，所有读写请求均被重定向至新存储（步骤 ②）；（3）清理阶段：垃圾回收完成后，系统安全地清除活动存储中的过期文件（步骤 ③-④）；（4）稳定状态：系统进入稳定状态，持续监控新存储的存储利用率及其他关键指标。

值得注意的是，该机制以迭代循环的方式运行：当新存储达到预设阈值或满足其他触发条件时，系统会启动新一轮垃圾回收。在后续的 GC周期中，当前的新存储将转变为压缩目标，其数据会被系统地合并到现有的已排序值日志中，同时建立一个新存储来处理传入的读写请求。这种循环设计在保持服务连续性的同时，确保系统存储效率得到持续优化。关于请求处理机制的更多细节将在第二节 D 中讨论。

阶段转换机制。GC 完成后，系统通过以下步骤从垃圾回收后阶段转换到下一周期的

算法1： 写操作处理程序

输入： (k, v)：要写入的键值对。
输出： 操作状态：成功或超时。

```
1: 函数 处理写入(k, v)
       阶段 1：日志条目创建与持久化
2:     entry ← 构造 Raft 日志条目(k, v)
3:     offset ←持久化到日志（当前日志， entry）
       阶段 2：共识与状态机应用
4:     t ← 初始化计时器（共识超时）
5:     result ← 等待提交或超时（entry， t）
6:     如果 result =已提交 则
7:         应用状态机（当前数据库， k， offset）
8:         返回 成功
9:     else
10:        返回 超时
11:    结束条件判断
12: 结束函数
```

算法 2： 并行点查询处理器

输入： k：查询键。
输出： 与 k 关联的值，或 NOT FOUND。

```
1：函数HANDLEREAD(k)
       阶段 1：GC 前状态
2:     如果 ¬GC Started则
3:         offset ←QUERYOFFSET(旧数据库， k)
4:         如果 offset = −1则返回 未找到
5:         结束如果
6:         返回 READVALUE(旧日志， offset)
7:     结束如果
       阶段 2：GC 期间状态
8:     如果 ¬GC已完成 则
9:         f_new-offset ← 异步查询偏移量(新数据库, k)
10:        f_old-offset ← 异步查询偏移量(旧数据库， k)
11:        offset ←AWAIT(f_new offset)
12:        如果 offset ≠ −1则
13:            返回 READVALUE(当前日志， offset)
14:        结束如果
15:        offset ←AWAIT(f_old offset)
16:        如果 offset = −1则 返回 未找到
17:        结束如果
18:        返回 READVALUE(旧日志， offset)
19:    结束如果
       阶段 3：垃圾回收后状态
20:    f_new-offset ← 异步查询偏移量(新数据库, k)
21:    f_sorted-value ← 异步查询值(sortedFile， k)
22:    offset ←等待(f_new offset)
23:    如果 offset ≠ −1则
24:        返回 READVALUE(当前日志， offset)
25:    结束如果
26:    value ← 等待(f_sorted value)
27:    如果 value = −1 则返回 未找到
28:    结束if
29:    返回 value
30: 结束函数
```

GC 前阶段：(1) 系统安全地移除过时的活动存储文件；(2) 随后，新存储在逻辑上成为下一周期的活动存储；(3) 重置 GC 状态标志，以指示处于 GC 前阶段。这种角色轮换可确保系统在多个 GC 周期内持续保持可用性。

D. 三阶段请求处理机制

为确保整个垃圾回收生命周期内请求处理正确，我们将请求处理机制划分为三个不同阶段：垃圾回收前、垃圾回收期间和垃圾回收后，如表 I 所示。本节重点介绍各阶段核心操作（写入、读取和扫描）的处理策略。

值得注意的是，读操作和写操作在不同 GC 阶段需要采用不同的处理策略。写操作（写入）与GC阶段无关：它们始终写入文件描述符当前引用的存储模块（当前日志和当前数据库），GC 触发时会自动切换这些模块。相比之下，读操作（读取和扫描）必须感知GC阶段，因为它们可能需要通过不同的查找路径查询多个存储模块，以获取最新数据。因此，我们先给出写入操作的统一算法，再针对读取和扫描进行分阶段分析。

1) 写入请求处理：算法1展示了一种经过优化的写操作机制，将高效的日志持久化与状态机应用相结合。该算法利用两个主要存储组件：当前日志 用于存储值，当前数据库 用于存储键-偏移量映射，同时通过共识超时 确保操作能够持续推进。

写操作分为两个截然不同的阶段：(1)日志条目创建与持久化（第 *2-3* 行）：系统首先构建一个包含键值对信息的 Raft 日志条目。随后，该条目会同步持久化到当前日志文件的磁盘中，并生成一个用于唯一标识值所在位置的偏移量。在我们的优化设计中，这个关键步骤是值唯一的一次写操作；(2)共识与状态机应用（第 *4-10* 行）：日志持久化后，系统会启动一个设有超时上限的共识过程（第 4-5 行）。通过 Raft 协议收到提交确认后（第 6 行），系统将该操作应用于状态机：只在存储引擎中存储轻量级偏移量，而非整个值（第 7 行）。

该设计通过键值分离实现了显著的效率提升：在创建日志条目期间仅持久化一次值，并在状态机中存储轻量级偏移量，系统将值写入次数从至少三次减少到仅一次。在保持 Raft 的强一致性保证的同时，这项优化对于大值操作尤其有效。

2）获取请求处理：算法 2 展示了不同阶段的点查询处理策略。该实现依赖几个关键的状态变量：*GC Started* 和*GC Completed*，表示垃圾回收阶段转换的布尔标志；*oldDB*和*newDB*，分别管理原始数据和新数据的键-偏移量映射的 RocksDB 实例；以及*oldLog* 和*currentLog*，分别表示存储未排序原始条目和新写入条目的 ValueLog 文件路径；还有*sortedFile*，表示垃圾回收完成后包含压缩后的历史数据的按值排序的文件。

点查询处理机制由三个

不同阶段：

- 垃圾回收前阶段（第 *2-7* 行）：垃圾回收开始前，系统仅查询活动存储，首先从旧数据库所引用的存储引擎文件中获取偏移量（第 3 行）。如果在存储引擎中找不到偏移量（即偏移量等于 -1），则判定该键未找到（第 4 行）；否则，从旧日志所引用的日志文件中获取相应的值（第 6 行）。

- 垃圾回收期间阶段（第 *8-19* 行）：系统通过并行查找来提升查询效率。系统同时发起两个针对键-偏移量映射的异步查询：一个查询新存储，另一个查询活动存储（第 9-10 行）。收到 NewStorage 返回的偏移量后，如果该偏移量存在，系统会立即从当前日志中获取相应的值（第 11-13 行）。这种优先处理方式确保能够访问最新的数据版本。只有在新存储中找不到该键时，系统才会使用活动存储返回的偏移量，从旧日志中获取值（第 15-18 行）。
- *GC* 后阶段（第 *20-29* 行）：系统同时向新存储发起异步偏移量查询，并直接向*sortedFile* 查询值（第 20-21 行）。如果在新存储中找到偏移量，系统会立即从当前日志 中获取值（第 22-24 行），因为其中保存的是最新数据。否则，系统将直接返回从*sortedFile*获取的值，其中以优化后的格式保存了所有历史数据（第 26-28 行）。

这种系统化的三阶段设计确保了存储结构优化过程中读取服务的持续可用性。通过精心设计的查找序列，系统兼顾了强一致性和高效的查询性能。

3) 扫描请求处理：算法 3 展示了不同阶段的范围查询处理方法。该机制将*sortedFile* 作为关键组件，在垃圾回收后保存排序数据条目，并支持 Raft 日志恢复，如第三节-C 所述。范围查询处理根据垃圾回收阶段分为三种不同场景：

- *GC* 前阶段（第 *2–4* 行）：由于垃圾回收尚未开始，系统只从活动存储查询数据（第 3 行）。

- *GC* 期间阶段（第 *5–7* 行）：垃圾回收期间，系统同时在活动存储和新存储模块中并行查询（第 5–6 行），因为新的传入请求会被重定向到新存储，而现有数据仍保留在活动存储中。
- *GC* 后阶段（第 *8–11* 行）：垃圾回收完成后，系统同时在最终压缩存储和新存储模块中并行查询（第 9–10 行），分别访问排序后的历史数据和新写入数据。

完成所有并行查询后，系统会合并

算法 3： 并行范围查询处理程序

输入： $[k_{start}, k_{end}]$：查询键范围。
输出： 范围内的键值对映射。
注意： 状态变量遵循与算法相同的语义
2.
1: **函数** HANDLERANGESCAN(k_{start}, k_{end})
阶段 1：GC 前状态
2: **if** ¬垃圾回收_已启动 **则**
3: **返回** SCANKEYS(旧数据库,旧日志, k_{start}, k_{end})
4: **结束如果**
阶段 2：GC 期间状态
5: **如果** ¬垃圾回收_已完成 **那么**
6: $f_{old} \leftarrow$ 异步扫描(旧数据库, 旧日志, k_{start}, k_{end})
7: $f_{new} \leftarrow$ A同步SCAN(新数据库, 当前日志, k_{start}, k_{end})
阶段 3：垃圾回收后状态
8: **else**
9: $f_{sorted} \leftarrow$ 异步排序扫描(*sortedFile*, k_{start}, k_{end})
10: $f_{new} \leftarrow$ ASYNCSCAN(新数据库, 当前日志, k_{start}, k_{end})
11: **结束 if**
12: 等待所有项(f_{sorted} 或 f_{old}, f_{new})
13: **返回** 合并结果(f_{sorted} 或 f_{old}, f_{new})
14: **结束函数**

这些查询结果采用基于版本的策略。在此合并过程中，新存储（以f_{new}表示）中的数据优先于旧文件或排序文件中的条目，从而确保保留最新的更新（第 12–13 行）。这种由 GC 状态驱动的三阶段并行查询设计，不仅保证了系统状态转换期间范围查询的正确性，还通过并行处理提升了查询效率。

E. 容错与安全性保证

容错。Nezha采用标准的Raft故障模型。在一个包含n个节点的集群中，系统最多可容忍 $\lfloor \frac{n-1}{2} \rfloor$ 个节点故障，同时保持安全性和活性。Nezha的节点故障处理机制与标准Raft基本相同，差异仅出现在垃圾回收操作期间。

垃圾回收期间发生故障时，恢复过程会先检查原子GC状态标志。如果垃圾回收尚未完成，系统会将排序文件中的最后一个键确定为GC中断点，并从该位置继续执行垃圾回收操作，从而确保垃圾回收操作的完整性。

***ValueLog* 一致性与恢复。**跨节点值日志同步可在垃圾回收期间发生结构变化时维持一致性。系统确保即使各节点上的物理值日志内容暂时不同，逻辑状态机仍保持一致。恢复利用最终压缩存储中的已排序值日志，作为高效的快照机制来实现跟随者追赶。

形式化安全验证。我们使用 TLA+ 规范对 Nezha 的安全属性进行了形式化验证，确认我们的修改保留了 Raft 的正确性保证。完整的形式化规范和证明可在我们的开源代码库中获取。

IV. 实现与评估

本节将首先在第 IV-A 节介绍 Nezha 的实现，然后在第 IV-B 节介绍实验设置。随后，我们将在第 IV-C 节详细评估不同值大小下的性能，在第 IV-D 节评估扫描基数与范围查询效率之间的关系，并在第 IV-E 节评估各种 Yahoo! 云服务基准测试（YCSB） [33] 工作负载模式下的系统行为。第 IV-F 节考察不同集群规模下的系统可扩展性，第IV-G节分析垃圾回收对系统性能的影响，第 IV-H 节评估不同 GC 状态下的恢复时间。

A. 系统实现

我们使用 Go（1.19.5 版本）将 Nezha 实现为一个原型系统。该实现采用 RocksDB 作为底层键值存储引擎，分布式通信框架则使用 gRPC [34] 并结合 Google Protocol Buffers [35] 构建，以实现节点之间高效的数据序列化与消息传递。所有节点均通过万兆以太网（10GbE）网络互联。我们的原型包含约 7.5K 行代码（不包括 gRPC 和 RocksDB）。源代码可在 https://github.com/Dshuishui/Nezha 获取。

B. 实验设置

环境与参数。 实验在一个由三个高 I/O 节点组成的集群上进行，每个节点均配备 Intel(R) Xeon(R) E5-2603 v3（12 个逻辑核心，2.4 GHz）、Ubuntu 20.04.4 LTS、64 GB 动态随机存取存储器和 2 TB 固态硬盘存储。

工作负载。 为系统地评估 Nezha 设计带来的性能优势，我们针对多种工作负载，从多个维度开展了广泛的对比实验。在基于值大小的性能分析中，每个键值对默认包含一个 10 B 的键和值大小不等的值（1 KB 至 256 KB），且在 Nezha 中每个键值对有三个副本。键的访问模式遵循 Zipf 分布。我们评估了 Nezha 的三种基本操作的性能：写入、读取和扫描。为更好地了解系统在实际场景中的行为，我们还根据表 II 所示的 YCSB 经典工作负载模式对 Nezha 进行了测试，这些模式用于模拟真实场景。

表 II 评估中使用的 YCSB 工作负载。

工作负载	写类型	查询类型	类别
Load	插入	/	仅插入
A	更新	点查询	50%写入 50%读取
B	更新	点查询	5%写入 95%读取
C	/	点查询	只读
D	插入	点查询	5%写入 95%读取
E	插入	范围查询	5%写入95%扫描
F	*RMW*	点查询	50%写入 50%读取

基线。 为全面评估 Nezha 的性能，我们选择了以下七种配置进行比较：

- *Original*：一种传统的分布式键值存储系统，将最先进的 Raft [15]共识与 RocksDB 存储引擎相结合。
- *PASV [28]*：一种适用于基于 LSM 树的关系型数据库的解决方案，通过移除存储引擎的预写日志来消除双重日志记录。
- *TiKV [31]*：一种企业级分布式键值存储系统，其架构与 Original 类似。
- *Dwisckey*：WiscKey 的分布式实现 [11]，继承了键值分离的设计理念，并将其扩展到分布式环境中。
- *LSM-Raft [30]*：这是一种近期提出的共识与存储协同设计方法，通过传输压实后的 SSTable，而非细粒度日志条目，减少跟随者侧的冗余压实。然而，在共识过程中，领导者仍会经历完全冗余的写入。
- *Nezha-NoGC*：Nezha 的基础版本，在 Raft 共识模块内实现键值分离。
- *Nezha*：Nezha 的完整系统，其垃圾回收机制与 Raft 的日志管理紧密耦合。

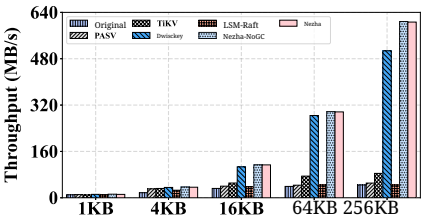
C. 基于值大小的性能分析

为验证键值分离对写操作性能的提升，以及适配 Raft 的垃圾回收机制对读取性能的平衡作用，我们从吞吐量和平均延迟两方面评估了写入、读取和扫描操作的性能。测试将键大小固定为 10 B，同时将值大小从 1 KB 逐步增加至 256 KB，以分析值大小对系统性能的影响。为评估基本操作性能，我们向系统加载了 100 GB 数据，并将活动存储模块的阈值设为 40 GB，以触发垃圾回收。

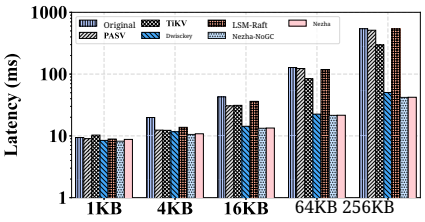
1) 写入性能比较：**吞吐量**。如图4 (a)所示，随着值大小从 1 KB 增至 256 KB，所有系统的吞吐量均呈上升趋势。值得注意的是，值越大，Nezha 和 Nezha-NoGC 的性能优势就越明显。这一趋势符合预期，因为在传统系统中，每个值都必须至少持久化三次（Raft日志、预写日志和 SSTable），较大的值会加剧写放大开销。相比之下，Nezha 的键值分离架构只将值持久化一次到值日志中，从而有效消除了冗余 I/O 开销。

平均而言，Nezha 的吞吐量比 Original 高 460.2%，Nezha-NoGC 的提升幅度相近，为 464.7%。Nezha 与 Nezha-NoGC 之间的差异极小，表明后台垃圾回收对写性能的影响微乎其微。

在其他基线系统中，PASV 通过消除存储引擎的预写日志，相比 Original 实现了适度提升（26.5%），但仍受 Raft日志和 SSTable 冗余的影响。作为一种分布式键值分离系统，Dwisckey 的表现

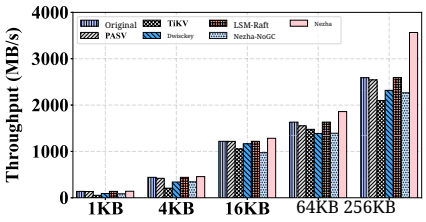


(a) 写入吞吐量

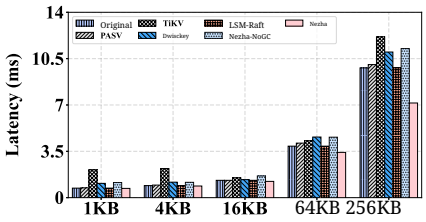


(b) 写入的平均写入延迟

图 4. 不同值大小下**写入请求**的性能比较。

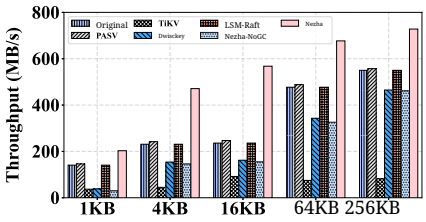


(a) 读取吞吐量

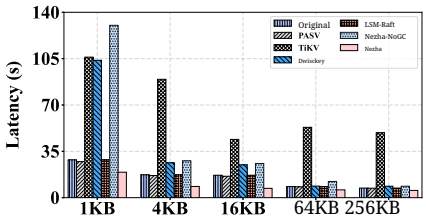


(b) 读取的平均读取延迟

图 5. 不同值大小下**点查询**的性能比较。



(a) 扫描的吞吐量



(b) 扫描的平均读取延迟

图6. 不同值大小下**范围查询**的性能比较；从100 GB数据集中查询4 GB数据。

接近 Nezha-NoGC，但由于额外的值持久化操作，性能略低（7.5%）。TiKV 是一款企业级系统，采用传统的 Raft-LSM 架构，因此表现出与 Original 类似的局限性。

LSM-Raft通过降低跟随者端的压实开销，性能比Original提升了16.9%。然而，Nezha的性能比LSM-Raft高出418%，因为关键写入路径主要受领导者而非跟随者的影响。Nezha通过减少领导者和跟随者上的冗余写入来解决这一瓶颈——将写入次数从至少三次降至恰好一次。

延迟。如图4(b)所示，延迟趋势与吞吐量观察结果一致。平均而言，与Original相比，Nezha将延迟降低了59.2%，而Nezha-NoGC的降幅相近，为60.76%。值得注意的是，尽管Nezha在加载100 GB数据的过程中执行了两次垃圾回收操作，其写延迟仍与Nezha-NoGC几乎相同，这表明后台垃圾回收对写性能的影响极小。第四节-G对垃圾回收的影响进行了详细分析。

2) 读取性能对比*:* **吞吐量**。获取操作的性能通过针对已加载的 100 GB 数据集发起 1,000,000 次点查询进行评估。如图5(a)所示，键值分离对读取性能呈现出双重影响。Nezha-NoGC 的吞吐量平均比 Original 低 21.3%，因为每次点查询在检索实际值之前都需要额外进行一次偏移量查找。这一开销在值较小时尤为明显，因为偏移量查找在总查询延迟中所占比例更大。

相比之下，Nezha 利用其垃圾回收机制，将数据重新组织为带有基于哈希的索引的排序文件，吞吐量比 Original 提高了 12.5%。哈希索引支持直接偏移量查找，有效弥补了键值分离固有的开销。值越大，这一优势就越显著，因为优化后的数据布局能带来更大的收益。

在其他基线中，PASV 和 LSM-Raft 的读取性能与 Original 相当，因为它们的优化侧重于写路径，并未修改读操作。Dwisckey 的性能与 Nezha-NoGC 相近，因为两者采用了类似的键值分离与数据检索机制。平均而言，Nezha 的性能比 Dwisckey 高出 37.34%，这表明 Nezha 的 GC 机制能够有效应对键值分离架构固有的读取性能挑战。

延迟。如图5 (b) 所示，延迟趋势与吞吐量观察结果一致。平均而言，由于额外的偏移查找开销，Nezha-NoGC 的延迟比 Original 高 28.7%，而 Nezha 通过基于哈希的索引实现直接偏移量查找，使延迟降低了 10.3%。

*3)*扫描性能比较：**吞吐量**。如图 6 (a) 所示，与点查询相比，键值分离对范围查询的影响更为显著。平均而言，Nezha-NoGC 的吞吐量比 Original 低 39.5%，明显差于点查询中观察到的 21.3% 降幅。这是因为范围查询需要检索多个连续的键值对，但键值分离会按到达顺序将值分散在值日志中，从而将顺序 I/O 转变为随机 I/O。

相比之下，Nezha 的吞吐量比 Original 提高了 72.6%。这一显著提升得益于 Nezha 的 GC 机制，该机制根据键顺序将分散的值重新组织到排序文件中。借助哈希索引，范围查询只需进行一次随机读取，即可定位已排序值日志中的起始位置，随后便可顺序读取后续键值对。这种方法比 Original 更高效，因为 Original 必须遍历多个 SSTable 文件

这些文件分布在不同的 LSM 树层级中。随着值大小增大，Nezha-NoGC 与 Nezha 之间的性能差距会缩小，因为单次 I/O 操作变大时，顺序 I/O 与随机 I/O 之间的差异也会减小。

在其他基线中，PASV 和 LSM-Raft 的扫描性能与 Original 相当，因为它们的写路径优化并未改善读操作的数据布局。Dwisckey 的性能与 Nezha-NoGC 相近，因为这两个系统都缺乏读优化数据重组。平均而言，Nezha 的性能比 Dwisckey 高出 208.9%，这表明 Nezha 的 GC 机制有效解决了键值分离通常带来的扫描性能下降问题。

延迟。如图 6 (b) 所示，延迟趋势与吞吐量观察结果一致。平均而言，由于分散的值带来的随机 I/O 开销，Nezha-NoGC 的延迟比 Original 增加了 107.2%，而 Nezha 通过恢复顺序访问模式的排序文件组织将延迟降低了 39.2%。这些结果证实，Nezha 有效解决了键值分离架构固有的扫描性能下降问题。

D. 基于扫描长度的性能分析

为评估扫描基数对范围查询性能的影响，我们进行了全面分析，测量了不同查询规模下的扫描吞吐量和延迟。我们系统地测试了分别检索 10、100、1000 和 10000 个键值对的范围查询，以了解各系统的可扩展性特征。在这些测试中，我们选择了 16 KB 的中间值大小，并使用 100 个并发线程对系统进行评估，与第四节-E中的 YCSB 工作负载*E* 配置保持一致。

吞吐量。如图 7 (a) 所示，本实验通过将检索的键值对数量从 10 调整至 10000，评估扫描基数对范围查询性能的影响。结果表明，在所有扫描长度下，Nezha 均持续优于 Original，平均吞吐量提升达到 7.58%。这一稳定的性能优势证实，无论扫描规模如何，Nezha 的 GC 机制都能为范围查询提供可靠的优化。其他基线的性能趋势与第四节-C中的观察结果一致，即 PASV 和 LSM-Raft 的性能与 Original 相近，而

Dwisckey 的表现与 Nezha-NoGC 相似。

延迟。如图 7（b）所示，延迟趋势与吞吐量的观察结果一致。Nezha 在所有扫描长度下都保持稳定且较低的延迟，而 Nezha-NoGC 由于缺少 GC 优化，键值分离带来的开销导致其延迟显著升高。这些结果进一步证实，Nezha 的垃圾回收机制有效解决了键值分离架构固有的读取性能下降问题。

*E. YCSB*工作负载性能

为验证系统在不同阶段、不同读写比例工作负载下的稳定性，并与所有基线系统比较性能

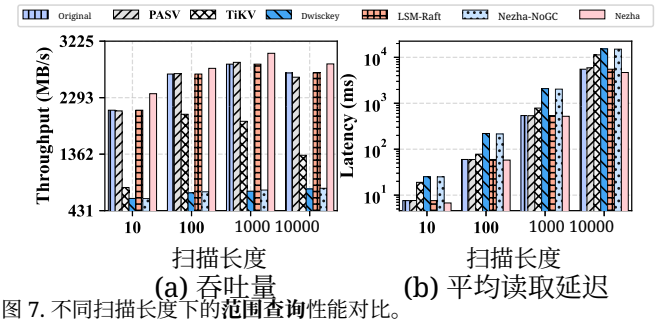


图 7. 不同扫描长度下的**范围查询**性能对比。

我们使用 YCSB 的六种经典工作负载（*A*-*F*）进行了全面测试。我们预加载了 100 GB 的随机数据。随后，针对每种工作负载类型执行了 1,000,000 个请求。为与之前的实验保持一致，所有工作负载的值大小均设为 16 KB。

吞吐量。如图 8（a）所示，本实验评估了系统在 YCSB 经典工作负载下的性能，这些工作负载模拟了具有不同比例读写操作的真实场景。结果表明，Nezha 在所有工作负载模式下的表现始终优于 Original，平均吞吐量提升 86.5%。

按工作负载类型分析，对于写密集型工作负载（*A* 和 *F* ，写入占比为 50%），Nezha 和 Nezha-NoGC 均因键值分离的优势而获得显著提升。对于读主导工作负载（*B*、*C*和*D* ，点查询占比为 95%–100%），Nezha 凭借其哈希索引有序文件保持了较强的性能，而 Nezha-NoGC 则因偏移查找开销而收益有限。对于扫描密集型工作负载 *E*，Nezha 实现了显著提升，而 Nezha-NoGC 则出现严重性能下降，这与第四节 C 节中的观察结果一致。

这些结果证实，Nezha 能够针对多种工作负载模式有效平衡读写性能，因此适用于读写操作混合的实际应用场景。其他基线的性能趋势与先前实验一致：PASV 和 LSM-Raft 的性能与 Original 相近，而 Dwisckey 的表现与 Nezha-NoGC 相似。

延迟。图8（b）和图8（c）分别展示了不同 YCSB 工作负载下的平均写延迟和读延迟。结果表明，在所有工作负载中，Nezha 的延迟始终低于基线系统。对于写操作，Nezha 显著降低了延迟，尤其是在写密集型工作负载中。对于读操作，Nezha 在各种工作负载模式中都保持了具有竞争力的延迟表现，包括极具挑战性的扫描密集型工作负载*E* ，而 Nezha-NoGC 在该工作负载下性能显著下降。这些观察结果证实，Nezha 能够在真实的混合工作负载下有效平衡写入与读取性能。

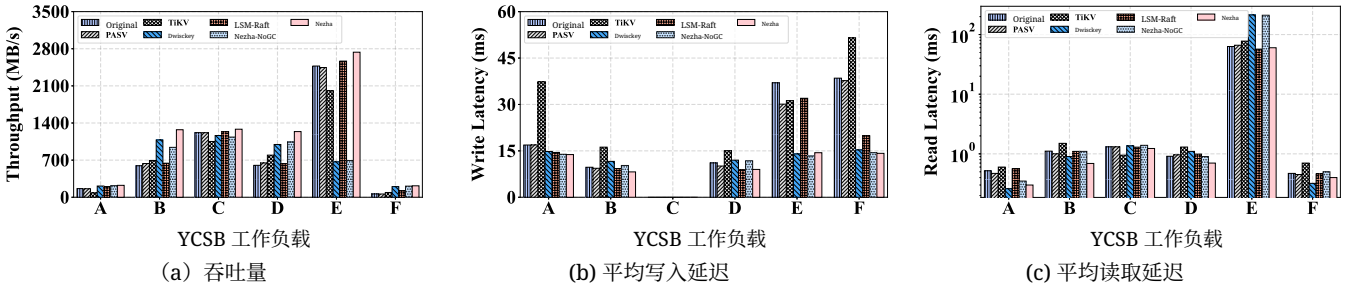


图 8. 吞吐量 与延迟比较：所有系统在 **YCSB** 工作负载下。

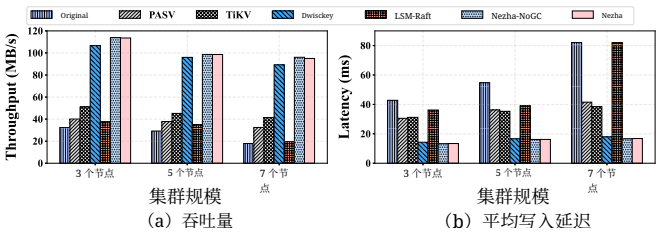


图 9. 不同集群规模下**写入** 请求的性能比较。

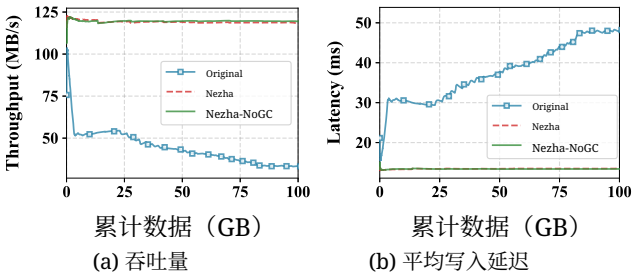


图 10. 垃圾回收对系统性能的影响。

F. 可扩展性分析

为了研究 Nezha 在更大规模集群配置下的可扩展性特征，我们分别在 3、5 和 7 节点集群上进行了实验。这些实验通过以固定值大小 16 KB 写入 100GB 数据来评估系统可扩展性，与我们之前的实验保持一致。

吞吐量与延迟。如图 9 所示，随着集群规模从 3 个节点增至 7 个节点，由于共识协调开销，所有系统的吞吐量均有所下降，延迟则有所增加。与不同集群规模下的 Original 相比，Nezha 展现出显著更强的可扩展性，吞吐量提高了 3.5× 至 5.3× 倍，延迟降低了 3.2× 至 4.9× 倍。这些结果证实，无论集群规模如何，Nezha 都能保持稳健的性能。

G. 垃圾回收对性能的影响

为量化垃圾回收机制对系统性能的具体影响，我们设计了一项针对性的长时间写入测试。在该实验中，我们将垃圾回收触发阈值设为 40 GB，并持续监测三个系统在写入 100 GB 数据过程中的性能：Original、Nezha 和 Nezha-NoGC。为与先前实验保持一致，所有基线系统的值大小均设为 16 KB，并每隔 100ms 收集一次性能快照。

吞吐量。如图 10(a) 所示，在写入 100 GB 数据的过程中触发了两次垃圾回收操作（分别在数据量达到 40 GB 和 80 GB 时）。从累计吞吐量曲线可以看出，尽管启用了垃圾回收机制，整个测试过程中 Nezha 与 Nezha-NoGC 的性能差距仍然很小，两者的吞吐量曲线几乎重合。这是因为触发垃圾回收时，Nezha 会将写请求原子地切换到新存储模块，而垃圾回收操作则异步地在独立的活动存储模块上执行，从而有效地将垃圾回收开销与关键写入路径解耦。相比之下，Original 系统的性能明显较差，这验证了键值分离架构在处理大规模数据写入方面的优越性。

延迟。图 10(b) 展示了数据写入过程中的延迟变化趋势。延迟分析证实了类似的发现：Nezha 和 Nezha-NoGC 的延迟性能几乎相同，且集中在较低范围；而 Original 系统的延迟分布更高、更分散。这种差异源于 Nezha 的键值分离架构，该架构降低了写放大，并避免了传统 LSM 树设计中由压实引起的延迟峰值。

H. 故障与恢复

为量化垃圾回收对故障恢复时间的影响，我们针对不同系统状态进行了恢复实验。图 11 展示了恢复时间对比：与 Original 系统相比，垃圾回收前、垃圾回收期间和垃圾回收后阶段的恢复时间分别缩短了 34.8%、34.5% 和 32.6%。这一改进归功于键值分离的引入：系统仅在 RocksDB 中存储轻量级偏移指针，而非完整值，从而显著减少了存储在其中的数据量。即使故障发生在垃圾回收期间，恢复过程也只需额外从已排序值日志中读取中断点，即可完成剩余的垃圾回收过程。

V. 相关工作

受共识与存储接口处性能瓶颈的启发，我们回顾了近期关于日志结构化键值存储、写放大降低以及共识与存储协同设计的研究，并讨论 Nezha 与现有方法的不同之处。

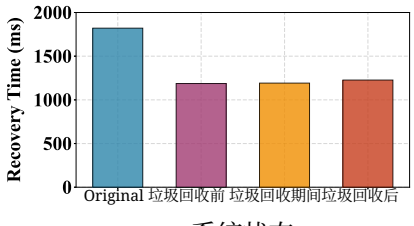


图 11. 不同系统状态的恢复时间比较。

A. 日志结构键值存储的最新进展

基于 LSM 树的键值存储凭借其写入优化设计，已成为现代存储系统的基础 [2]–[4]。近期研究着重于提升 I/O 效率，并使 LSM树适应新兴硬件和架构范式。在硬件方面，研究人员致力于通过生命周期感知型数据管理，使 LSM 树适配 ZNS 固态硬盘，从而优化数据放置并降低维护开销 [36], ；利用 GPU 卸载来加速性能关键操作 [37],; 通过自适应主机-CSD 卸载适配计算存储驱动器 [38], ；以及借助大容量动态随机存取存储器和内存高效索引 [39]。在架构方面，云原生部署需求推动了针对解耦内存 [40],、无服务器环境 [41],、多租户约束下数据机密性 [42], ，以及将云存储视为主要持久化层的基于对象存储的“无盘”设计 [43] 的适配工作。此外，基于学习的方法 [44], [45] 能够在动态工作负载下实现自适应参数调优。

然而，这些工作主要优化 LSM 树内部机制，以适应不断演进的系统特性与部署环境，而并未系统地重构其与共识协议的交互。相比之下，Nezha 通过共识与存储协同设计应对跨层冗余，将键值分离直接集成到 Raft 协议中，从而消除冗余持久化。由于 Nezha 运行于共识层且不修改 LSM 树内部机制，这些硬件与架构适配仍然正交于我们的方法，并可与 Nezha 结合以进一步提升性能。

B. 写放大降低技术

除了硬件和架构适配之外，写放大仍然是基于 LSM 树的存储系统面临的一项根本性性能挑战。针对这一问题，研究者从两个互补的方向进行了探索。WiscKey [11], 开创的键值分离，通过仅在 LSM 树中存储键、将值放入独立的仅追加日志中，减少写放大，从而避免在压实过程中反复移动较大的值。压实是写放大的另一个主要来源，也因此受到广泛关注，相关优化包括工作负载感知型调优[46],、硬件卸载 [47]–[49], 、流水线并行[50],、资源感知型调度 [51], [52], 以及文件选择优化 [53]。

然而，这些工作仅在单节点存储引擎内部优化写放大。Nezha 则在一个根本不同的层面上应对写放大：它将键值分离直接集成到 Raft 共识层，从而减少整个分布式集群中的冗余持久化操作——将操作次数从至少三次降至仅一次。由于 Nezha 在更高的抽象层级上运行，且不修改 LSM 树内部机制，这些单节点优化与我们的方法彼此正交。

C. 共识与存储协同设计及优化

随着研究人员认识到冗余持久化操作带来的性能开销，共识协议与存储系统的协同设计日益受到关注。TiKV [31],、CockroachDB [18],和 PolarDB [20] 等生产系统采用传统的分层架构，其中基于 Raft 的共识与存储引擎彼此独立运行，导致多个层之间存在冗余持久化。近期研究通过消除系统各层之间的冗余日志记录 [28], [29] ，以及优化共识协议机制以缩短关键路径 [54],[55], 来解决这一问题。LSM-Raft [30] 则兼顾这两个方向：它传输压缩后的 SSTable，而非细粒度条目，从而减少从节点侧冗余写入。

然而，LSM-Raft 的优化仅惠及跟随者，因为领导者仍需承受完整的冗余写入。Nezha 更进一步，通过将键值分离与 Raft 在架构层面集成，进一步消除了领导者侧的冗余。PASV [28] 和 PALF [29] 等先前工作至多减少一次持久化操作，而 Nezha 将值持久化次数从至少三次降至仅一次，同时保留 Raft 安全性属性。此外，Nezha 还可以与这些方法结合，进一步降低键的持久化开销。

VI. 结论

本文介绍了 Nezha，这是一种新颖的分布式键值存储系统，通过深度集成共识层与存储层，实现了全面的性能优化。Nezha 的核心是 KVS-Raft，这是一种创新的共识协议，将键值分离理念融入 Raft 共识层，大幅减少了整个分布式集群中的存储冗余和写入开销。为充分发挥系统的潜力，我们设计了一种 Raft 感知型垃圾回收框架，在保持强一致性保证的同时优化读取性能。这些架构创新，加上三阶段请求处理机制，使 Nezha 相较于传统的基于 Raft 的存储系统取得了显著的性能提升，并通过广泛的实验评估得到验证。

致谢

本研究得到中国国家重点研发计划（2022YFB4501703）、江西省职业早期青年科技人才培养项目（20252BEJ730003）和江西省自然科学基金（20252BAC200615）的资助。

参考文献

[1] (2025) LevelDB。 [在线]。网址：https://github.com/google/leveldb [2] (2025) RocksDB。 [在线]。网址：https://rocksdb.org/ [3] F. Chang、J. Dean、S. Ghemawat、W. C. Hsieh、D. A. Wallach、M. Burrows、T. Chandra、A. Fikes 和 R. E. Gruber，“Bigtable：一种用于结构化数据的分布式存储系统”，《ACM计算机系统汇刊》（TOCS），第26卷，第2期，第1–26页，2008年。[4] G. DeCandia、D. Hastorun、M. Jampani、G. Kakulapati、A. Lakshman、A. Pilchin、S. Sivasubramanian、P. Vosshall 和 W. Vogels，“Dynamo：亚马逊的高可用键值存储”，《ACM SIGOPS操作系统评论》，第41卷，第6期，第205–220页，2007年。[5] K. Srivastava 和 N. Shekokar，“面向电子商务业务模式的多语言持久化方法”，载于《2016年信息科学国际会议》（ICIS）。IEEE，2016年，第7–11页。[6] J. M. Pujol、V. Erramilli、G. Siganos、X. Yang、N. Laoutaris、P. Chhabra 和 P. Rodriguez，“能够扩展在线社交网络的小型引擎”，《IEEE/ACM网络汇刊》，第20卷，第4期，第1162–1175页，2012年。[7] Y. Wang、J. Yuan、S. Wu、H. Liu、J. Chen、C. Ma 和 J. Qin，“LeaderKV：通过学习型索引和解耦键值表提升键值存储的读取性能”，载于《2024年IEEE第40届数据工程国际会议》（ICDE）。IEEE，2024年，第29–41页。[8] P. O’Neil、E. Cheng、D. Gawlick 和 E. O’Neil，“日志结构合并树（LSM 树）”，《Acta Informatica》，第33卷，385，1996年4月，第351页。[9] A. Lakshman 和 P. Malik，“Cassandra：一种去中心化结构化存储系统”，《ACM SIGOPS操作系统评论》，第44卷，第2期，第35–40页，2010年。[10] (2025) Apache HBase。 [在线]。网址：https://hbase.apache.org/ [11] L. Lu、T. S. Pillai、H. Gopalakrishnan、A. C. Arpaci-Dusseau 和 R. H. Arpaci-Dusseau，“WiscKey：在面向固态硬盘的存储中分离键和值”，《ACM存储汇刊》（TOS），第13卷，第1期，第1–28页，2017年。[12] X. Wu、Y. Xu、Z. Shao 和 S. Jiang，“LSM-Trie：一种基于 LSM 树、面向小型数据项的超大型键值存储”，载于《2015年USENIX年度技术会议》（USENIX ATC 15），2015年，第71–82页。[13] L. Lamport，“Paxos 简明教程”，《ACM SIGACT新闻·分布式计算专栏》，第32卷，第4期，第51–58页，2001年。[14] ——，“兼职议会”，载于《并发：Leslie Lamport文集》，L. Lamport 编。ACM，2019年，第277–317页。[15] D. Ongaro 和 J. Ousterhout，“探寻易于理解的共识算法”，载于《2014年USENIX年度技术会议》（USENIX ATC 14），2014年，第305–319页。[16] D. Ongaro，《共识：连接理论与实践》。斯坦福大学，2014年。[17] (2025) etcd。 [在线]。网址：https://github.com/etcd-io/etcd [18] R. Taft、I. Sharif、A. Matei、N. VanBenschoten、J. Lewis、T. Grieger、K. Niemi、A. Woods、A. Birzin、R. Poss 等，“CockroachDB：具备弹性的地理分布式SQL数据库”，载于《2020年ACM SIGMOD数据管理国际会议论文集》，2020年，第1493–1509页。[19] D. Huang、Q. Liu、Q. Cui、Z. Fang、X. Ma、F. Xu、L. Shen、L. Tang、Y. Zhou、M. Huang 等，“TiDB：一种基于 Raft 的 HTAP数据库”，《VLDB基金会会刊》，第13卷，第12期，第3072–3084页，2020年。[20] W. Cao、Z. Liu、P. Wang、S. Chen、C. Zhu、S. Zheng、Y. Wang 和 G. Ma，“PolarFS：一种面向共享存储云数据库、具有超低延迟和故障恢复能力的分布式文件系统”，《VLDB基金会会刊》，第11卷，第12期，第1849–1862页，2018年。[21] Y. Xiao、N. Zhang、W. Lou 和 Y. T. Hou，“区块链网络分布式共识协议综述”，《IEEE通信调查与教程》，第22卷，第2期，第1432–1465页，2020年。[22] B. Cao、Y. Li、L. Zhang、L. Zhang、S. Mumtaz、Z. Zhou 和 M. Peng，“当物联网遇上区块链：分布式共识面临的挑战”，《IEEE网络》，第33卷，第6期，第133–139页，2019年。[23] S. Li、G. Oikonomou、T. Tryfonas、T. M. Chen 和 L. Da Xu，“一种用于面向服务的物联网决策的分布式共识算法”，《IEEE工业信息学汇刊》，第10卷，第2期，第1461–1468页，2014年。[24] N. T. Nguyen，“分布式时态数据库中的基于共识的时间戳”，《计算机杂志》，第44卷，第5期，第398–409页，2001年。[25] Q. Cai、W. Guo、H. Zhang、D. Agrawal、G. Chen、B. C. Ooi、K.-L. Tan、Y. M. Teo 和 S. Wang，“高效的分布式内存管理借助远程直接内存访问和缓存，”VLDB基金会会议录，第11卷，第11期，第1604–1617页，2018年。[26] Y. Oh、J. Choi、D. Lee 和 S. H. Noh，“少用缓存，性能更佳：在混合存储系统中平衡缓存大小与闪存缓存的更新成本。”载于 FAST，第12卷，2012年。[27] R. F. Freitas 和 W. W. Wilcke，“存储级内存：下一代存储系统技术”，《IBM研究与发展杂志》，第52卷，第4.5期，第439–447页，2008年。[28] K. Huang、Z. Shen、Z. Jia、Z. Shao 和 F. Chen，“在基于LSM树的关系数据库中通过被动数据持久化消除双重日志记录”，载于第20届USENIX文件与存储技术会议（FAST 22），2022年，第101–116页。[29] F. Han、H. Liu、B. Chen、D. Jia、J. Zhou、X. Teng、C. Yang、H. Xi、W. Tian、S. Tao 等，“Palf：面向分布式数据库的复制式预写日志”，VLDB基金会会议录，第17卷，第12期，第3745–3758页，2024年。[30] X. Zhang、X. Tan、S. Song、X. Huang 和 J. Wang，“LSM-Raft：针对LSM树存储优化Raft”，ACM数据管理会议录，第3卷，第6期，第1–27页，2025年。[31] (2025) TiKV。 [在线]。可访问：https://github.com/tikv/tikv [32] D. Ongaro 和 J. Ousterhout，“探寻易于理解的共识算法（扩展版）。Raft共识算法。2014年。”[33] B. F. Cooper、A. Silberstein、E. Tam、R. Ramakrishnan 和 R. Sears，“使用YCSB对云服务系统进行基准测试”，载于第1届ACM云计算研讨会会议录，2010年，第143–154页。[34] (2025) gRPC。 [在线]。可访问：https://grpc.io/ [35] (2025) Google Protocol Buffers。 [在线]。可访问：https://github.com/protocolbuffers/protobuf [36] G. Liu、C. Yang、Q. Yu、C. Guo、W. Xia 和 Z. Cao，“Prophet：通过文件生命周期预测和压实补偿优化ZNS固态硬盘上的基于LSM的键值存储”，载于第38届国际大规模存储系统与技术会议，2024年。[37] S. Pandey 和 A. Basu，“H-rocks：基于持久内存的CPU-GPU加速异构RocksDB”，ACM数据管理会议录，第3卷，第1期，第1–28页，2025年。[38] S. Nie、C. Zhang、M. Li、F. Yu、Y. Li 和 W. Wu，“Zoomdb：在ZNS固态硬盘和叠瓦式磁记录硬盘上构建高性价比键值存储引擎”，《系统架构杂志》，第103465页，2025年。[39] H. Sun、D. Kong、S. Jiang、Y. Yue 和 X. Qin，“Triekv：以内存为一等公民的高性能键值存储设计”，《IEEE并行与分布式系统汇刊》，2024年。[40] R. Wang、J. Wang、P. Kadam、M. T. ¨Ozsu 和 W. G. Aref，“dlsm：一种面向内存解聚的基于LSM的索引”，载于2023年IEEE第39届数据工程国际会议（ICDE）。IEEE，2023年，第2835–2849页。[41] M. Liu、J. Gu 和 T. Zhao，“Serverlesslsm：为弹性无服务器LSM树键值存储而设计的高速远程直接内存访问协同解聚压实”，载于2025年IEEE国际Web服务会议（ICWS）。IEEE，2025年，第1–7页。[42] V. Thakkar、D. Kim、Y. Lai、H. Kim 和 Z. Cao，“Shield：对LSM键值存储中的持久化数据进行加密，适用于从单体存储到解聚存储的场景”，ACM数据管理会议录，第3卷，第3期，第1–28页，2025年。[43] (2025) SlateDB。 [在线]。可访问：https://github.com/slatedb/slatedb [44] W. Yu、S. Luo、Z. Yu 和 G. Cong，“Camal：通过主动学习优化LSM树”，ACM数据管理会议录，第2卷，第4期，第1–26页，2024年。[45] J. Liu、F. Wang、D. Mo 和 S. Luo，“结构设计与最优性相遇：在庞大的配置空间中探索优化的LSM树结构”，ACM数据管理会议录，第2卷，第3期，第1–26页，2024年。[46] A. Huynh、H. A. Chaudhari、E. Terzi 和 M. Athanassoulis，“探索LSM树的灵活性与鲁棒性”，《VLDB期刊》，第33卷，第4期，第1105–1128页，2024年。[47] C. Ding、J. Zhou、K. Lu、S. Li、Y. Xiong、J. Wan 和 L. Zhan，“D2comp：利用解聚存储中的数据处理单元高效卸载LSM树压实”，《ACM体系结构与代码优化汇刊》，第21卷，第3期，第1–22页，2024年。[48] C. Ding、K. Lu、Q. Zhang、Z. Ye、T. Yao、D. Wang、H. Wu 和 J. Wan，“Dflush：面向解聚的基于LSM的键值存储，由数据处理单元卸载刷新操作”，ACM数据管理会议录，第3卷，第3期，第1–28页，2025年。[49] D. Tang、W. Wang、Y. Mao、J. Yu、T.-W. Kuo 和 C. J. Xue，“Stem：针对LSM中大规模压实的基于流处理的FPGA加速

键值存储”，载于*2024*年*IEEE*第*40*届数据工程国际会议（*ICDE*）。IEEE，2024年，第3893–3905页。[50] H. Byun、H. Yoo 和 S. Park，“重新审视LSM树中的多线程压实：实现压实流水线”，载于第*54*届并行处理国际会议，2025年，第794–803页。[51] Q. Yu、C. Guo、J. Zhuang、V. Thakkar、J. Wang 和 Z. Cao，“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”，*ACM*数据管理会议录，第2卷，第3期，第1–28页，2024年。[52] H. Wang、J. Qiu、F. Yuan 和 H. Zhang，“重新思考LSM树中的压实策略”，*ACM*数据管理会议录，第3卷，第3期，第1–26页，2025年。[53] R. Wei、Z. Zhu、A. Kryczka、J. Zhuang 和 M. Athanassoulis，“对RocksDB中部分压实的写放大进行基准测试、分析与优化”，载于第*28*届数据库技术扩展国际会议（*EDBT 2025*）。OpenProceedings.org，2025年，第425–437页。

[54] Y. Wang、Z. Wang、Y. Chai 和 X. Wang，《重新思考分布式系统中 Raft 的线性一致性约束》，*IEEE*知识与数据工程汇刊，第35卷，2023年11月，829、11815页。[55] Y. Xu、H. Zhu、P. Pandey、A. Conway、R. Johnson、A. Ganesan 和 R. Alagappan，《Ionia：面向现代磁盘键值存储的高性能复制》，载于第*22*届*USENIX*文件与存储技术会议（*FAST 24*），2024年，第225–241页。