计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(3): 550−566, 2026

DPZB+tree: 基于 ZNS SSD 与持久化内存的高效 B+树索引设计

曹 夕 李明杰 杨朝树 杨 程 张润宇
(公共大数据国家重点实验室(贵州大学) 贵阳 550025)
(贵州大学计算机科学与技术学院 贵阳 550025)
(gs.ycao23@gzu.edu.cn)

DPZB+tree: An Efficient B+tree Index Design for ZNS SSD with Persistent Memory

Cao Xi, Li Mingjie, Yang Chaoshu, Yang Cheng, and Zhang Runyu
(*State Key Laboratory of Public Big Data* (*Guizhou University*), *Guiyang* 550025)
(*College of Computer Science and Technology*, *Guizhou University*, *Guiyang* 550025)

Abstract Emerging zoned namespace solid state drive (ZNS SSD) addresses several critical issues inherent in traditional SSDs, including high write amplification, low storage density, and complex I/O paths. These advancements create new opportunities for the progress of storage technologies. The B+tree, as an efficient tree index structure, is widely used in various databases and file systems to support the efficient loading of large models, the construction of external knowledge bases, and the management of structured metadata, thereby significantly improving training efficiency and knowledge invocation performance. However, due to the hardware characteristics of ZNS SSDs being different from traditional block devices, directly deploying B+tree to ZNS SSDs not only results in higher write amplification rates but also causes cascading updates, seriously affecting the performance of the storage system. For the above problems, we propose DPZB+tree, a B+tree indexing structure optimized for ZNS SSDs and incorporating persistent memory (PM). First, DPZB+tree adopts a hybrid DRAM-PM-ZNS SSD storage architecture, thereby effectively separating hot data and cold data. Second, DPZB+tree introduces a lightweight hot/cold node identification strategy, which in turn improves I/O efficiency. Third, to address the limited capacity of PM, we present a dynamic node placement strategy that adaptively migrates data between PM and ZNS SSD. Finally, we combine hardware characteristics and spatiotemporal locality principles to design leaf node splitting and merging operations. DPZB+tree is implemented using a ZNS SSD simulator and Intel optane PM. Experimental results show that under various workloads, DPZB+tree outperforms LSM tree, SSDB+tree, DZB+tree, and Baseline in terms of performance and recovery time.

Key words zoned namespace solid state drive (ZNS SSD); persistent memory; B+tree; index structure; temperature aware

摘 要 新型分区命名空间固态硬盘(zoned namespace solid state drive, ZNS SSD)有望解决传统块设备固态盘写放大率高、存储密度低、I/O路径复杂等问题,为存储技术的发展创造机遇。B+树作为一种高效的树形索引结构,被广泛应用于各类数据库和文件系统中,以支撑大模型高效数据加载、外部知识库构建及

收稿日期: 2025-06-17; **修回日期**: 2025-12-15
基金项目: 国家自然科学基金项目(62362009, 62562014, 62162011); 贵州大学引进人才科研项目(2022[44]); 贵州省科技厅支持省外高水平大学帮扶贵州高校学科建设科研项目(黔科合人才 XKBF[2025]017)
This work was supported by the National Natural Science Foundation of China (62362009, 62562014, 62162011), the Doctoral Funds of Guizhou University (2022[44]), and the Guizhou Provincial Department of Science and Technology Foundation (XKBF[2025]017).
通信作者: 张润宇(zhangry@gzu.edu.cn)

结构化元数据管理，从而显著提升训练效率与知识调用性能。然而，由于 ZNS SSD 的硬件特性不同于传统块设备，直接将 B+树部署到 ZNS SSD 中不仅会导致较高的写放大率，还会引起级联更新，严重影响存储系统的性能。针对以上问题，结合新型持久化内存（persistent memory, PM）提出了一种基于 ZNS SSD 的 B+树索引结构 DPZB+tree。首先，DPZB+tree 采用 DRAM-PM-ZNS SSD 混合存储架构，实现冷热数据分离存储；其次，DPZB+tree 设计了冷热节点识别策略，以提高存储系统的读写效率；然后，针对 PM 容量有限的问题，提出了冷热节点动态放置策略，实现冷热数据的自适应迁移；最后，结合硬件特性和局部性原理设计了叶节点分裂及合并操作。DPZB+tree 索引方案基于 ZNS SSD 模拟器和英特尔傲腾 PM 实现。实验结果表明，在多种工作负载下，相较于 LSM-tree, SSDB+tree, DZB+tree, Baseline, DPZB+tree 均取得了优异的读写性能以及更低的恢复耗时。

关键词 分区命名空间固态硬盘；持久化内存；B+树；索引结构；温度感知

中图法分类号 TP311.13

DOI: 10.7544/issn1000-1239.202550474 **CSTR:** 32373.14.issn1000-1239.202550474

随着云计算、大数据、物联网以及人工智能等新一代信息技术的发展，全球数据量急剧增长。据国际数据公司预测，2028 年全球数据量将增长至 393.8 ZB（1ZB≈10^{12}GB），相比 2018 年增长 9.8 倍[1]。数据的激增对存储系统的时效性、可靠性以及容量提出了更高的要求[2]。因此，传统的机械硬盘逐渐无法满足大数据时代的高性能存储需求，正被速度快、体积小及抗震耐用的闪存固态硬盘取代[3]。与传统机械硬盘不同，闪存固态盘具有其独特的物理特性：闪存单元遵循先擦除后写入的原则，并通过地址重映射机制实现覆盖写入[4]。因此，为了向上层系统提供类似于机械硬盘的块设备访问通道，闪存固态盘内部集成了闪存转换层[5-7]（flash translation layer, FTL），用于将逻辑地址转换为物理地址，同时负责垃圾回收、坏块替换和磨损均衡等管理任务。近年来，为降低大容量闪存管理开销，研究者进一步提出了分区命名空间固态硬盘[8-10]（zoned namespace solid state drive, ZNS SSD）并迅速成为学术界与工业界的研究热点[11-12]。与传统基于块设备接口的固态硬盘不同，ZNS SSD 将设备内部的垃圾回收等数据管理操作上移到主机端，从而避免预留冗余空间，并有效缓解传统闪存设备在垃圾回收时不可预测的性能下降问题。

B+树[13]是一种从 B 树[14]演化来的支持多路查找的树形索引结构。由于 B+树具有高度的平衡性和高效的查找性能，B+树索引结构被广泛地用于各类数据库（如 MySQL[15]）和文件系统（如 NTFS[16] 和 XFS[17]）中。此外，在大模型训练过程中，需从数据库高效加载数据，而 B+树作为核心索引结构，通过有序性和范围扫描能力实现训练样本的快速定位，能显著缩短模型训练时间[18]。在检索增强生成系统中，B+树还可管理知识库的结构化元数据，通过筛选低维特征快速缩小向量索引的检索范围，实现高效知识调用。B+树的确定性索引与向量索引的近似能力分层协同，有望成为提升训练效率与检索性能的关键支撑。然而，现有的 B+树索引结构大多数是基于 FTL 层设计的。如果不加修改地将基于 FTL 设计的 B+树索引结构部署到 ZNS SSD 上，不仅会导致过高的写回开销，还会在 ZNS SSD 分区中产生较多碎片，频繁地触发垃圾回收操作，从而影响存储系统性能。

近年来，还涌现出诸多的新型持久化内存（persistent memory, PM）。这些 PM 具有非易失、可按字节寻址、高存储密度和低延迟等优点[19-20]。因此，本文引入可字节寻址和支持就地更新的 PM 作为存储系统的中间层，开展面向 ZNS SSD 的 B+树索引结构优化研究。然而，受限于原理、工艺以及成本等现实因素，STT-RAM 等 PM 技术路线存储密度较低，其容量与外存仍有数量级差距[21]。因此，本文从 B+树的访存行为模式出发，充分考虑各存储介质特性，利用大容量 ZNS SSD 存储冷叶节点、小容量 PM 存储少量热叶节点以及伪叶节点，以阻断索引结构链式更新，避免由小粒度随机写入导致的写放大问题，实现索引结构的合理组织，提高系统的整体吞吐量。

本文的主要贡献有 3 个方面：

1）提出了基于 DRAM-PM-ZNS SSD 混合存储架构的 B+树索引结构 DPZB+tree，充分利用不同存储介质的特性，实现 B+树节点的合理组织以及分类管理，以阻断级联更新，降低写放大率，避免性能抖动。

2）设计了冷热节点动态管理策略，结合轻量级访问时序分析，实现热数据自适应迁移与冷数据高效替换，提高小容量 PM 缓存命中率，降低写入开销。

3)基于傲腾 PM[22] 和 NVMeVirt[23] 模拟器构建了实验平台。实验结果表明，本文提出的方案在读写性能、尾延迟以及断电恢复时间等方面均优于对比对象。此外，本文方案的相关代码已经发布在 GitHub① 上，以供参考。

1 背景与相关工作

1.1 分区命名空间固态硬盘

近年来，随着闪存芯片容量的激增，采用块设备接口来管理闪存的方式暴露了诸多问题。首先，闪存的读写粒度与擦除粒度不一致。为了保持数据一致性，固态硬盘在执行垃圾回收时需要将有效数据迁移到预留的额外空间，这一机制导致了 7%~28% 的空间浪费[24]。其次，为了支持块设备的就地修改功能，固态硬盘内部需要配备大容量的 DRAM 用于存储映射表。最后，闪存块设备固有的垃圾回收操作增加了写放大效应，还可能引发不可预测的长尾延迟，严重影响系统性能。随着全球数据量的快速增长，传统块设备闪存的局限性愈发明显，逐渐难以满足大数据时代对高性能和低能耗存储系统的需求。

为此，研究学者提出了 ZNS SSD。与传统块设备 SSD 相比，ZNS SSD 通过分区存储模型[25] 有效解决了以上问题。其顺序写入机制减少了写放大效应，降低了垃圾回收的开销，同时提高了存储空间利用率。在主机直接参与数据管理的模式下，ZNS SSD 避免了传统固态盘 FTL 层的冗余数据迁移，大幅提升了设备的写入性能和使用寿命，同时显著降低了功耗。顺序写入和分区管理还减少了资源冲突和不可预测的长尾延迟，使系统性能更加稳定和可靠。目前，国内外各存储设备服务商相继推出了 ZNS SSD 产品。例如，2020 年，西部数据研发出 96 层 3D TLC NAND 闪存的 ZNS SSD[26]；2021 年，三星电子推出了支持分区命名空间技术的 PM1731a[11] 等。ZNS 技术标准已被正式纳入了非易失性存储器(non-volatile memory express, NVMe)2.0 协议[27]，同时 Linux 内核也新增了对 ZNS 设备的支持[28]。因此，基于 ZNS SSD 的存储系统架构已经成为未来关键发展方向。

1.2 闪存索引结构

日志结构合并树(log-structured merge-tree, LSM-tree)[29] 是一种被广泛应用的键值索引结构。由于 ZNS SSD 的特性高度契合 LSM-tree 的数据存储和管理方式，基于 ZNS SSD 的 LSM-tree 优化研究得到了广泛的关注。RocksDB[30] 通过 ZenFS[24] 将 LSM-tree 存储到 ZNS SSD。近年来还有研究团队针对 LSM-tree 的合并及垃圾回收过程，取得了压缩感知分区分配算法 CAZA[31]、高效键值存储方法 ZNSKV[32] 及动态区域映射算法 Brick-ZNS[33] 等研究成果。然而，现有基于 ZNS SSD 的 LSM-tree 研究仍较少关注其固有的读放大率高、查询性能低等缺陷。

相比之下，B+树具备平衡且高效的查找性能。因其特性，B+树尤其适用于那些涉及大量检索、范围查询和顺序遍历的应用场景。B+树的叶节点按顺序链接，所有实际数据仅存储在叶节点中，而非内部节点。这一设计使得 B+树在进行范围查找时能够通过顺序扫描叶节点链表快速获取数据。此外，相比 B 树，B+树具有较低的高度，减少了查找操作的磁盘 I/O 次数，提升了查询性能。

B+树的基本操作主要包括读、插入、删除和范围查找。其中，读操作通过从根节点开始逐层向下查找，最终定位到叶节点。插入操作首先需要找到合适的位置进行插入。如果目标叶节点未满，数据将直接插入；若叶节点已满，则需要进行节点分裂，并将新键值向上传递。为了保持树的平衡，这一操作可能引发父节点的级联更新。删除操作在执行时会在节点中删除目标键值。如果在删除后节点中的键值数量低于给定阈值，则会触发节点合并操作。范围查找操作通过顺序遍历叶节点链表来查找符合条件的数据。

近年来，随着闪存技术的快速发展，众多学者针对闪存设备和 B+树的特性展开研究。2020 年，Liang 等人[34] 针对固态混合硬盘写放大高、寿命有限以及随机访问性能差等问题提出了 XsB+tree。其通过将内部节点和冷数据存储于 SSD，叶节点和热数据放置在叠瓦式磁记录的热区，减少了写入放大，延长了 SSD 的使用寿命，提升了存储系统的访问性能。

2021 年，Do 等人[35] 针对基于传统固态硬盘的日志技术开销大的问题，提出了一种新的批量 I/O 存储接口。通过将日志结构化操作迁移到 SSD 控制器中，并设计新的 FTL 层以及垃圾回收机制，显著提升了 BW-tree 在 SSD 上的性能，保障了存储成本与系统性能的均衡。

2023 年，刘扬等人[36] 提出了 ZNS SSD 感知的索引结构 ZB+-tree，将索引节点分散存储于 ZNS SSD 的

① https://github.com/GZUsys/DPZB-tree

传统分区和顺序分区中，并设计不同节点结构来适应各类分区的写入特性。此外，通过对节点采用不同的控制合并策略，在保证索引性能的同时尽量减少级联更新次数。然而，由于 SSD 本身的读写粒度与索引结构更新粒度的差异较大，仍难以避免较高的闪存页面写放大率。

2023 年，Duffy 等人[37]针对键值固态硬盘（key-value SSD, KVSSD）性能较低以及缺乏高效索引等问题，提出了一种键值存储系统 Dotori。Dotori 设计了一种专用于 KVSSD 的 B+树，通过 KVSSD 提供的接口管理 B+树节点、节点动态分裂和合并等策略，以降低地址映射开销。然而，KVSSD 需要特殊硬件支持，无法直接适配 ZNS SSD。

1.3 新型 PM

近年来，还涌现了诸多新型 PM，包括相变存储器[38]、电阻式存储器[39]、铁电存储器[40]和自旋转移力矩存储器[41]等。PM 具有可按字节寻址、非易失、低延迟、低静态功耗和高存储密度等优点。具体来说，PM 结合了传统 DRAM 内存和外存块设备的优势。与 DRAM 相比，PM 不仅具有更大的容量与更低的单位成本，还提供了与外存块设备类似的持久性，即断电后数据不会丢失。而与外存块设备相比，PM 则拥有较高的访问速度和较低的访问延迟，使其在提供持久存储的同时，仍能保持接近内存的性能。PM 的独特优势使其在高性能计算和大数据处理等应用中展现出巨大的潜力。

本文利用 PM 可字节寻址和就地更新等优势，开展面向 ZNS SSD 的 B+树索引结构优化研究，旨在减少直接迁移 B+树到 ZNS SSD 上带来的频繁小粒度写入，并消除级联更新，从而实现基于 ZNS SSD 的高性能 B+树。

2 研究动机

数据的爆发式增长对存储系统的性能、功耗以及每位存储成本提出了更高要求。特别是物联网设备产生的时序数据，这类数据按时间顺序产生且通常具有数据规模大、热数据访问操作多和写多读少等特点。这不仅要求数据存储的成本要尽可能的低，同时还要求对数据的冷热进行区分，对热数据的操作要尽可能的友好。而 ZNS SSD 作为新兴的低成本大容量存储设备，有望成为其有效的数据存储解决方案。但是，由于闪存介质固有的异地更新限制，现有面向块设备的 B+树索引部署到 ZNS SSD 上往往会带来严重的级联更新问题。因为缺乏 FTL 层，叶节点的更新通常会级联到根节点。而在具有兄弟节点指针的 B+树中，当更新最右边的叶节点时，这种更新甚至可能级联到整个树结构，从而带来巨大的写放大开销。

为了展示基于块设备的 B+树迁移到 ZNS SSD 上的性能，本文对比了 B+树在传统 SSD 和 ZNS SSD 上的性能特征，结果如图 1 所示。实验所用的服务器操作系统为 Ubuntu 22.04.3, Linux 内核版本为 5.15.0。本文使用 NVMeVirt[22]模拟 ZNS SSD 设备，并使用由雅虎公司开源的数据库基准测试工具（yahoo! cloud serving benchmark, YCSB）[42]生成服从 Zipfian 分布的工作负载进行测试。

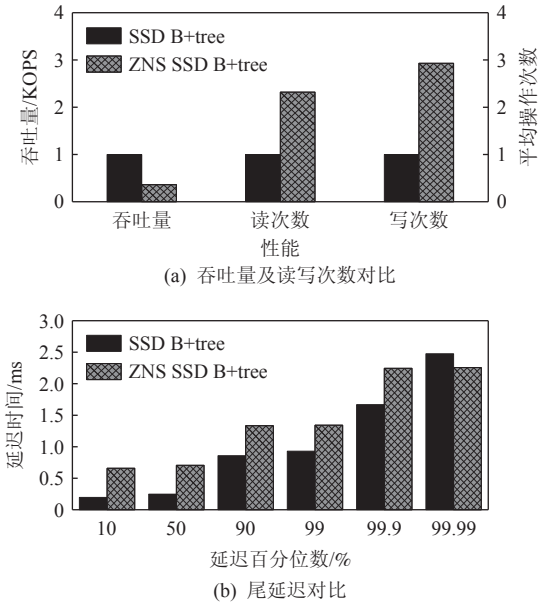


Fig. 1 Performance characteristics comparison of B+trees

图 1 B+树性能特征对比

图 1(a)反映了二者关于吞吐量、读次数和写次数的性能。实验证明，不加任何修改地将 B+树迁移到 ZNS SSD 上会造成严重的写放大。B+树在 ZNS SSD 上的吞吐量仅为 SSD 上的 40%，且读次数增加了 2.3 倍，写次数增加了近 3 倍，符合预期结果。

此外，本文还比较了尾延迟，结果如图 1(b)所示。10%~99.9% 尾延迟在 SSD 上的 B+树表现都优于 ZNS SSD 上的 B+树，而在 99.99% 延迟处 SSD 上的 B+树性能较差，这是由于传统 SSD 设备的固有缺陷。为了向主机提供类似于块设备接口，SSD 使用 FTL 隐藏了 SSD 设备的内部特性。

综合分析以上实验结果可以得知，ZNS SSD 可以避免传统块接口设备垃圾回收时不可预测的长尾

延迟问题。即使在发生级联更新的情况下, ZNS SSD 上的 99.99% 尾延迟也低于 SSD 上的尾延迟。但是, 不加任何修改地直接将 B+树迁移到 ZNS SSD 会带来严重的写放大开销和级联更新开销, 严重影响存储系统的性能。因此, 亟需开展面向 ZNS SSD 的 B+树索引结构优化研究, 以提升系统存储性能, 实现对数据的优化存储和高效访问。

3 DPZB+tree 设计

3.1 总体设计

本文提出了一种基于 DRAM-PM-ZNS SSD 混合存储架构的温度感知的 B+树索引结构 DPZB+tree, 其节点分为 3 层, 节点结构如图 2 所示。为提升 B+树的操作效率、重构效率和系统资源利用率, 本文采用了内部节点结合伪叶节点和冷热节点的设计方案, 其中冷热节点依据最近访问时间进行区分。不同类型的节点根据存储介质的特点分别存储: 内部节点数据存储在 DRAM 上, 伪叶节点和热节点数据存储在 PM 上, 冷节点数据存储在 ZNS SSD 上。该架构充分利用了 ZNS SSD 的顺序写入特性、PM 的非易失性存储优势以及 DRAM 的高速访问性能。

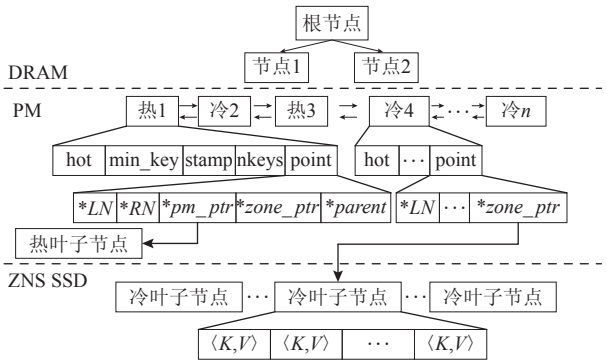


Fig. 2 Schematic diagram of DPZB+tree node structure

图 2 DPZB+tree 节点结构示意图

针对内部节点, DPZB+tree 将其存储在 DRAM 中。首先, DRAM 提供了比 PM 和 SSD 更快的读写速度, 这样可以在查找节点时实现更高效的数据访问。其次, DRAM 具有可就地更新的特性, 这一特点能够有效阻断级联更新, 避免了传统 B+树中节点间的递归更新, 从而大幅降低写放大率。此外, 由于内部节点仅存储索引信息而不包含实际的数据, 即使因系统断电或发生故障导致内部节点丢失, DPZB+tree 也能快速依靠存储在 PM 中的伪叶节点重新构建内部节点。这样既可以确保对内部节点执行操作时的高效性, 又提供了容错能力, 提升了系统的可靠性。

针对叶节点的存储, 本文采用了 2 种不同的策略。首先, 叶节点的关键索引信息只存储在 PM 中, 本文将其称为伪叶节点。这样做的目的是确保在系统断电或崩溃时, 能够从第 1 个伪叶节点重构整棵树, 保证数据的可靠性与恢复能力。同时, 这种做法避免了对 PM 的过度占用, 平衡了系统的存储效率和资源消耗。其次, 为区分冷热节点, 本文记录最近访问时间, 将时间最近的密集访问叶节点划分为热节点, 其余节点划分为冷节点。热节点存储在 PM 中, 从而高效地执行读写操作, 有效防止频繁的小粒度更新导致 ZNS SSD 上的写放大和垃圾回收问题。冷节点则被存储在 ZNS SSD 上, 因为这些数据不需要频繁更新, 且 ZNS SSD 适合存储大容量的冷数据, 能够在减少写放大的同时提供更高的存储密度, 降低存储成本。

具体来说, 本文的伪叶节点由以下部分组成: 节点最小值、热节点标志位、父节点指针、关键字数量、左兄弟节点指针、右兄弟节点指针、指向 PM 的指针、指向 ZNS SSD 的指针以及访问时间戳。节点最小值用于重构 B+树, 在系统正常重启时, 通过节点最小值可以快速构建内部节点, 避免进行全盘扫描, 从而提升系统恢复效率; 热节点标志位用于标记叶节点是否为热叶节点; 父节点指针用于记录叶节点在 B+树内的父节点位置信息, 通过父节点指针可以方便地向上寻找父节点的位置; 关键字数量用于记录该叶节点所包含的键值对的数量; 左兄弟节点指针和右兄弟节点指针用于连接同一层级的相邻叶节点, 使得叶节点之间形成有序的双链表结构, DPZB+tree 可以通过这些指针快速遍历相邻的叶节点; PM 指针在叶节点为热节点时生效, 指向 PM 中存储的实际键值对, 以支持热数据的快速读写, 此时 ZNS SSD 指针无效; ZNS SSD 指针在叶节点为冷节点时生效, 指向 ZNS SSD 中存储的实际键值对, 用于访问大容量冷数据, 此时 PM 指针无效; 访问时间戳代表了节点最近一次进行写入的时间, 用于冷热节点替换算法。

对于存储在 PM 中的节点, 为利用 PM 设备的非易失性, 在修改节点后会对该节点进行持久化操作, 保证发生掉电或系统崩溃等情况时的数据一致性。特别是当发生冷热节点动态替换时, 伪叶子节点中指向实际存储位置的 PM 指针或者 ZNS SSD 指针会在执行完迁移算法后再进行更新并执行持久化操作, 以保证 PM 和 ZNS SSD 之间的数据一致性。

DPZB+tree 采用 DRAM-PM-ZNS SSD 混合存储架构的设计, 充分利用不同存储介质的特性, 实现对

数据的合理布局和高效访问。DPZB+tree 主要包含 3 个关键技术。首先，DPZB+tree 利用低开销 1 维 K 均值（K-means）算法识别当前热写入请求，并通过建立缓存节点访问时间戳进行冷热分离，以解决写入频繁的节点带来的写放大问题。其次，DPZB+tree 设计了基于叶节点访问时间戳的缓存节点替换算法，以在剩余 PM 空间不足时替换最不活跃节点，保证 PM 缓存当前最热的节点。最后，DPZB+tree 充分利用 PM 特性并基于时空局部性设计了分裂和合并策略，降低了写放大率。下面将详细介绍这 3 种关键技术。

3.2 写入请求冷热识别策略

DPZB+tree 采用了 DRAM-PM-ZNS SSD 三级存储架构，其中 PM 用于存储热数据节点，ZNS SSD 则用于存储冷数据节点。冷热数据分布规律普遍存在于各类读写密集型应用系统中。在典型的读写密集型应用场景下，数据访问请求往往呈现出特定的统计分布特征，即大部分操作集中在小部分数据上。实现冷热数据的精准识别与合理分布是系统设计的关键所在，这不仅能够充分发挥 DPZB+tree 多级存储架构的优势，更能显著提升整体存储系统的性能表现。

1 维 K-means 算法是聚类算法 K-means 在 1 维数据空间中的特例，用于处理单维度数据的聚类问题。其目标是将 1 维数据集划分为 K 个簇，使得每个簇内的数据点尽可能接近其质心，同时使得不同簇之间的数据点尽可能分离。在实际应用中，1 维 K-means 算法能够在少量迭代内快速收敛，尤其适用于处理写入序列等简单场景，为数据布局优化提供了关键支持。

基于此，本文提出的 DPZB+tree 采用低开销 1 维 K-means 算法对写入请求进行冷热识别。在执行读写操作之前，DPZB+tree 会对当前接收到的所有读写操作请求对应的键值进行聚类分析，通过 K-means 算法计算位于聚类中心的键值。如果该键值到达其他键值的平均距离最短，表明该键值对应的节点预计将承担后续大部分的读写请求，因此被识别为当前读写序列的热点键值。基于这一预测，DPZB+tree 在执行实际读写操作之前，会将该键值对应的节点动态迁移至 PM 中（若该键值对应的节点当前存储于 ZNS SSD）。从而优化热数据的访问性能，有效降低存储系统的响应延迟，进而提升系统的整体读写效率。基于请求序列聚类的热访问请求识别流程如图 3 所示。

其具体步骤如下：首先，基于时空局部性原理，

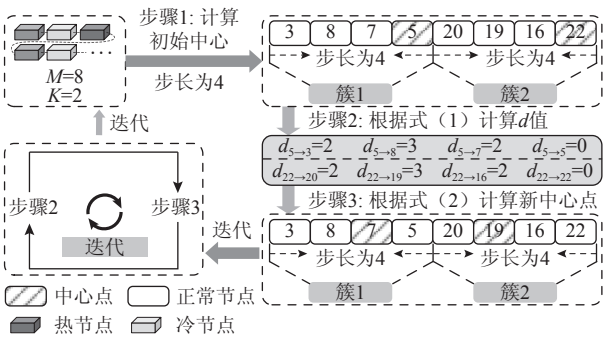


Fig. 3 Schematic diagram of hot node identification

图 3 热节点识别示意图

算法从 M 个当前读写请求中以 M/K 为步长选取 K 个请求作为初始中心点，并依据就近原则将每个读写请求分配到键值差距最小的中心点所属聚类中，从而形成初始聚类集合 G。以图 3 中所示的 8 个读写请求为例，DPZB+tree 将读写请求划分为 2 个聚类，其聚类中心分别是 5 和 22。然后根据式（1）计算各请求与中心点之间的键值差距 d。键值差距定义为各键值与中心键值的绝对值。接着，根据式（2）计算当前各聚类的新中心点，并基于该点重新划分聚类。随后，算法迭代上述 2 步操作，直至新旧中心点键值差距小于给定阈值 δ。此时，找到的 K 个键值即为当前写入序列的热键值，其对应的 K 个节点即为当前热写入节点。

$$d\left(k_i,k_j\right)=|k_i-k_j|,i=1,2,\cdots,M,j=1,2,\cdots,K。\quad(1)$$

$$k_j=\overline{k_i},k_i\in G_j。\quad(2)$$

此外，需要额外说明的是，通过计算得到的均值可能与实际写入序列中的键值不完全一致，但因为在计算均值时 DPZB+tree 采用了取整操作，确保了结果为整数，所以 DPZB+tree 始终能够定位到其对应的节点。同时，B+树的内部节点采用范围索引机制，例如，若内部节点为 [5, 14, 19]，则对应 4 个叶节点，其键值范围分别为 [0, 4], [5, 13, 14, 18], [19, +∞)。因此，无论聚类得到的键值是多少，均能通过范围匹配定位到对应的热叶节点。

算法 1 是该策略的操作流程。如果 B+树为空，将直接在 PM 创建第 1 个叶节点，并将当前请求序列都写入 PM。若 B+树不为空，首先通过 1 维 K-means 算法识别热点键值，并调用 B+树的查询操作定位这些键值对应的叶节点。随后，通过叶节点的热节点标志位判断其是否已经作为热节点存储在 PM。若其不在，则要将其从放置冷数据的 ZNS SSD 中迁移到放置热数据的 PM 中。在此过程中，若 PM 空间不

足,则需先执行热节点替换算法,选择最近访问时间戳最远的热节点写回 ZNS SSD,以释放空间供新热节点使用。通过以上步骤识别出当前写入序列的 K 个热键值后,此时这些键值对应的叶节点即为当前写入序列的热节点。在执行具体操作之前,需要确保这些叶节点已经作为热节点存储在 PM 上。这一机制确保了对热数据的高效存储与访问,同时优化了存储资源的利用率。

算法 1. 热节点处理算法。

输入: *buffer*[*n*](读写请求序列);

输出: 将热键值对应的热节点迁移到 ZNS SSD。

① if tree is null then
② *CreateFirstNode*();
③ else
④ *hot_key* = *K_means*(*buffer*[*n*]);
⑤ *cur_node* = *Search*(*hot_key*);
⑥ if *cur_node* 是冷节点 then
⑦ if PM 中没有存储热节点的空间 then
⑧ *HotNodeReplace*(*cur_node*);
⑨ else
⑩ 迁移 *cur_node* 到 PM 中;
⑪ 将 *cur_node* 添加到动态热节点数组中;
⑫ end if
⑬ end if
⑭ end if

3.3 冷热节点动态放置策略

PM 的访问速度接近 DRAM,同时具备非易失性。本文采用 PM 作为存放 B+树热节点的高速缓存区,通过设计高效的数据管理策略,使 PM 优先存储频繁访问的热数据,在有限 PM 容量下最大化提升热点访问性能。同时,将低频访问的冷数据迁移至 ZNS SSD,从而充分发挥分层存储的优势。

基于上述分析,本文采用小容量 PM 存储频繁访问的热数据,并设计了一种基于访问时间戳的冷热节点动态放置策略,以实现数据在 PM 与 ZNS SSD 之间的高效迁移,从而确保热节点的有效性以及对小容量 PM 的高效利用。当缓存的热节点数量达到上限时,部分热节点可能因访问频率下降而逐渐变为冷节点。由于 PM 的缓存能力有限,如果这些冷节点继续占用宝贵的 PM 空间,会显著影响存储系统的整体性能。因此,本文提出的动态放置策略通过实时监控节点的访问模式,可以动态调整数据分布,以优化存储资源利用率并提升系统性能。

具体来说,本文在伪叶节点层维护了叶节点的访问时间戳来作为判断叶节点冷热的依据。在初始状态下,PM 中不存在热叶节点,即热叶节点集合 H 为空集。每当有新的缓存节点 n 被添加进集合 H(PM 未满时在 PM 中创建新节点)时或叶节点 n 被执行写操作时,其访问时间戳 T_a 被置为当前时间戳 T_c。当集合 H 元素个数达到上限时,PM 无法容纳新的节点。此时,系统会依据叶节点的访问时间戳执行冷热节点替换。

冷热节点替换操作如图 4 所示。对于写入序列来说,本文通过写入请求冷热识别策略识别出 K 个写入热请求。这 K 个热请求对应的 K 个节点在接下来的写入中会承担大部分的写入,因此为了减少因频繁写入 ZNS SSD 而导致的写入放大和垃圾回收开销,DPZB+tree 需要动态地将这 K 个节点中尚未作为热节点存储在 PM 的部分迁移至 PM 中。

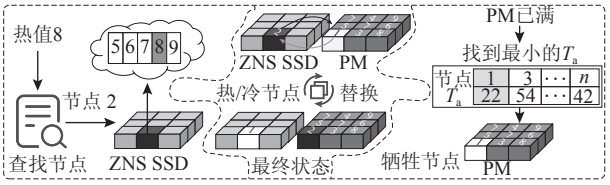


Fig. 4 Schematic diagram of hot node and cold node replacement

图 4 冷热节点替换示意图

首先,DPZB+tree 识别出热值为 8,并通过查找发现其所在的节点 2 作为冷节点存储在 ZNS SSD 中。因此,在执行写入操作前,需将节点 2 整体迁移至 PM 中作为热节点。然而,此时 PM 已满,因此需要遍历热节点指针动态数组,找到访问时间戳离当前时间戳最远的叶节点,即最冷的节点。如式(3)所示,DPZB+tree 通过计算 T_c-T_a 的值,选择差值最大的节点作为待替换的冷节点。

$$\max_{n}(T_c - T_a), n \in H。 \tag{3}$$

在本例中,节点 1 被选为待替换节点。随后,将节点 2 迁移至 PM,并更新其访问时间戳为当前时间戳,同时将节点 1 写回 ZNS SSD。通过这一机制,DPZB+tree 能够确保 PM 中 B+树节点的合理组织,有效降低节点写放大率。具体操作流程如算法 2 所示。

算法 2. 热节点处理算法。

输入: *cur_node*(待处理冷节点);

输出: 将待处理冷节点从 ZNS SSD 迁移到 PM,并将被替换的热节点从 PM 迁移到 ZNS SSD。

① *victim_node* = *FindMinStampHotNode*();
② *pos* = append *victim_node.K_V* to ZNS SSD;

③ *victim_node.zone_ptr* = *pos*;
④ *free*(*victim_node.pm_ptr*);
⑤ *K_V* = new *AllocKeyValue*();
⑥ *K_V* = read from ZNS SSD using *cur_ptr*;
⑦ *new_K_V* = *AllocKV*();
⑧ *copy*(*K*, *V*) from *K_V* to *new_K_V*;
⑨ *persist*(*new_K_V*);
⑩ *cur_node.pm_ptr* = *new_K_V*;
⑪ *cur_node.hot* = true;
⑫ *cur_node.stamp* = *current_time*;
⑬ *persist*(*cur_node.pm_ptr*);
⑭ *persist*(*cur_node*);
⑮ add *cur_node* to the dynamic hot node array;
⑯ *free*(*K_V*)。

3.4 分裂合并策略

B+树的叶节点分裂和合并操作在保证树的平衡、优化存储空间利用率、提高查询和写入效率、减少磁盘 I/O 次数以及提升系统稳定性等方面起到了至关重要的作用。尤其在需要处理大规模数据、频繁更新和查询的应用场景中，良好的分裂和合并策略能够显著提升存储系统的整体性能。

针对节点分裂操作，本文考虑到时空局部性原理，需确保待分裂的叶节点无论是冷节点还是热节点，在分裂后都应作为热节点存储在 PM 中。这是因为节点分裂通常意味着已经发生了大量的写入操作。

在初始情况下，DPZB+tree 写入的第 1 条数据对应的节点将作为热节点存储在 PM 上。同时，在 PM 存储足够新热节点的情况下，当发生节点分裂，分裂完的新节点应作为热节点存储在 PM 上。当 PM 的存储空间已满时，若发生节点分裂，则分为 2 种情况。1）针对热节点的分裂，DPZB+tree 仅需找到 1 个存储在 PM 且访问时间戳最远的热节点，将其刷回到 ZNS SSD，以释放 PM 空间。然后，将待分裂热节点的一半键值写入到在该空闲空间中新创建的热叶节点内，并修改叶节点元数据信息。同时，将新节点的首地址加入到热节点动态数组中，以便后续对其进行访问。2）而对于冷节点的分裂，DPZB+tree 则需要找到具有最远和次远访问时间戳的 2 个热节点，将它们刷回 ZNS SSD。算法 3 描述了节点分裂的详细步骤。

算法 3. 叶节点分裂算法。

输入：*leaf_node*（待分裂节点）；

输出：将待分裂节点分裂为 2 个热节点。

① *newleaf* = *AllocLN*();
② *N* = *leaf_node.nkeys*;
③ if 如果叶节点为热节点 then
④ if PM 中没有存储热节点的空间 then
⑤ 删除访问时间戳最远的热节点；
⑥ end if
⑦ 迁移 *leaf_node* 部分键值对到 *newleaf*;
⑧ *persist*(*leaf_node*);
⑨ *persist*(*newleaf*);
⑩ 将 *newleaf* 添加到热节点动态数组中；
⑪ else
⑫ if PM 中没有存储 2 个热节点的空间 then
⑬ 删除访问时间戳为最远和次远的热节点；
⑭ end if
⑮ *newleaf_*1 = *AllocLN*();
⑯ 迁移 *leaf_node* 部分键值对到 *newleaf_*1;
⑰ 迁移 *leaf_node* 剩下键值对到 *newleaf*;
⑱ *persist*(*newleaf*);
⑲ *persist*(*newleaf_*1);
⑳ 将 *leaf_node* 添加到热节点动态数组中；
㉑ 将 *newleaf* 添加到热节点动态数组中；
㉒ *free*(*leaf_node*);
㉓ end if
㉔ *Insert_Into_Parent*(*newleaf*)。

对于合并操作，DPZB+tree 的核心设计目标是最大化 PM 的利用效率。当叶节点在执行完删除操作后，如果其键值对数量小于最大存储容量的一半，则会触发节点合并操作。在合并过程中，优先选择左右兄弟节点中的热节点进行合并，以充分利用 PM 中的热节点资源，从而提升合并后的访问效率和存储性能。合并操作的具体规则如下。

1）特殊情况处理。①如果目标叶节点是第 1 个叶节点（即无左兄弟节点），则直接与右兄弟节点合并。②如果目标叶节点是最后 1 个叶节点（即无右兄弟节点），则直接与左兄弟节点合并。

2）兄弟节点选择策略。①如果左右兄弟节点同时存在且冷热状态不同，则优先与热节点合并。②如果左右兄弟节点同时存在且冷热状态相同，则默认与左兄弟节点合并。

3）父节点调整。合并操作完成后，父节点的子节点数量减 1。如果此时父节点的子节点数量小于其最大存储容量的一半，则会触发父节点的调整操作：①若兄弟节点的子节点数量大于最小容量要求，则从兄弟节点中迁移 1 个子节点。②若兄弟节点不可迁移，则与兄弟节点合并。

DPZB+tree 通过上述分裂与合并策略，不仅维护

了 B+树的动态平衡，还优化了 PM 的空间使用效率，从而实现了性能与资源利用的平衡。在分裂操作中，DPZB+tree 通过动态调整节点分布，提升了存储系统的读写性能；在合并操作中，DPZB+tree 保证待合并节点优先与热节点合并，进一步提升了 PM 的利用率，减少了缓存空间浪费。

3.5 数据一致性讨论

在混合存储架构中，PM 作为热数据缓存层，其数据持久性与 ZNS SSD 的写入原子性需协同保障。对于 PM 中的节点更新操作，我们采用有序持久化策略，首先完成节点内键值数据的写入与持久化操作，然后更新元数据中的键值数据字段并将其持久化。此顺序确保了即使发生系统崩溃，未完成更新的元数据也不会指向部分写入的键值数据，从而避免脏数据残留。此外，在冷热节点动态替换时，由于 PM 上的指针均为 8B，保障了支持原子性就地更新。在 ZNS SSD 上，所有键值数据均以追加方式顺序写入。仅当数据成功写入且持久化后，才更新 PM 中对应的元数据。由于 ZNS SSD 写指针具有单调递增特性，未提交的写入操作不会覆盖已提交区域。因此，ZNS SSD 上的写操作不会产生数据不一致性问题。

为实现系统异常掉电后的状态恢复，本文设计了多层级恢复流程。首先，通过 PM 中持久化存储的首节点指针定位首个伪叶子节点，并通过该节点依次访问整个伪叶子节点层；然后，遍历伪叶子节点层，检测其中“已写入 PM/ZNS SSD 但未更新元数据”的未完成操作节点，并清除这些不一致数据；最后，基于 PM 中伪叶子节点的完整元数据信息，重构 DRAM 中的 B+树内部节点索引结构，从而完成恢复流程。

4 实验结果及分析

4.1 实验对比对象介绍

目前，学术界尚未开展面向 DRAM-PM-ZNS SSD 的多级索引架构相关研究。由于写时复制 B+树(CoWB+tree)是一种采用追加写机制的 B+树结构，其数据写入方式符合 ZNS SSD 的区域顺序写入限制，特性与 ZNS SSD 具有较高的相似性。因此，本文将其作为研究的对比基准并对其进行了适应性改造。

具体而言，本文将 CoWB+tree 的内部节点和叶节点均存储在 ZNS SSD 上。当对 CoWB+tree 的叶节点执行写操作时，系统无法直接就地修改，而是将更新后的叶节点写入到对应区域的当前写指针位置。由于叶节点地址发生改变，必须同步更新父节点中指向子节点的指针。这种级联更新机制会逐级向上传播直至根节点，导致 CoWB+tree 的写入操作会产生与树高成正比的写放大效应，从而显著影响系统性能。鉴于这一特性，本文适应性地改造了 CoWB+tree，使其能够在 ZNS SSD 上运行。由此，本文通过 CoWB+tree 验证 DPZB+tree 在阻断级联更新方面的有效性。

此外，为了评估基于 PM 缓存的热节点策略的实际效果，本文进一步对 CoWB+tree 的架构做了改进，将其调整为 DRAM-ZNS SSD 两级存储结构的 B+树，并命名为 DZB+tree。DZB+tree 采用 DRAM 存储内部节点、ZNS SSD 存储叶节点的分层存储方案。在更新叶节点时，得益于 DRAM 支持就地更新的特性，级联更新过程将被有效阻断，从而显著优化了更新性能。

为了更全面地评估 ZNS SSD 相较于传统 SSD 的性能优势，本文还引入了基于传统 SSD 的 B+树作为对比对象，并将其命名为 SSDB+tree。SSDB+tree 的内部节点和叶节点均存储在传统 SSD 上，得益于传统 SSD 支持就地更新的特性，其更新操作可以直接在原有地址上完成。此外，由于传统 SSD 采用逻辑地址到物理地址的映射表 FTL 管理数据存储，当叶节点发生更新时，仅需更新映射表中的地址映射关系，而无需像 CoWB+tree 那样逐级更新父节点指针，从而有效阻断了级联更新的传播，这使得 SSDB+tree 在更新操作中避免了级联更新带来的写放大问题。因此，SSDB+tree 也为本文提供了一个重要的性能对比基准，以更清晰地揭示 ZNS SSD 在特定场景下的优势。

为了验证本文提出的 DBZP+tree 在读取和范围查找方面相较于 LSM-tree 的优势，本文还将其与基于 ZNS SSD 并使用 ZenFS 作为存储后端文件系统的 LSM-tree 进行了对比。

4.2 实验配置与测试负载

本文采用 NVMeVirt 作为工具来模拟 ZNS SSD 设备。作为目前较新且被广泛应用的闪存设备模拟器，NVMeVirt 具有执行速度快、支持接口广以及可扩展性高等特点。NVMeVirt 在 PCI 层进行设备模拟，支持通过 libzbd 库访问 ZNS SSD 设备。与较早的 ZNS 模拟方案 null_blk[43] 以及 FEMU[44] 相比，它能提供更精准的模拟访问延时。此外，NVMeVirt 还提供了丰富的可配置参数，如 NAND 闪存通道数、通道的逻辑单元数以及通道带宽等，从而保障实验结果的真实性和可靠性。实验使用 gcc 11.4.0 编译器构建 NVMeVirt，并借助西部数据的 ZBD 设备操作库 Libzbd 对 ZNS SSD 进行相应的操作。实验环境配置如表 1 所示。

Table 1 Experimental Platform Configuration

表 1 实验平台配置

配置类型	参数
操作系统	Ubuntu 22.04.3 LTS
内核版本	5.15.0-105-generic
处理器	Intel Xeon® Gold 6330 (28 core, 2.00 GHz)
DRAM	DDR4 256 GB
PM	4×128 GB Optane DC PM Module
编译环境	GCC/G++ 11.4.0
Libzbd	2.0.4

本文利用 NVMeVirt 创建了 50 个 2 GB 大小的顺序写入区域，用于模拟 ZNS SSD 设备。闪存页大小与读写延迟均保持默认设置。本实验仅使用 1/10 000 ZNS SSD 总容量（约 10MB）的 PM 空间缓存热节点，并在 4.4 节测试了 DPZB+tree 在 PM 使用量为 2~20 MB 的性能变化。

为全面评估本文提出的实验方案及对比对象的性能，本文采用 YCSB 生成了符合 Zipfian 分布的工作负载。其中包括单一工作负载和混合工作负载，工作负载配置如表 2 所示。在每个测试中，实验对象首先加载 300 万条操作使系统达到稳定状态，随后进行正式测试，以确保获取的性能指标具有较高的准确性。所有工作负载的测试条目均由 8B 的键和 8B 的值组成，这是 B+树实验的典型配置。

Table 2 Workloads Configuration

表 2 工作负载配置

工作负载	操作	分布
INSERT-1	仅插入	Zipfian
READ-1	仅读取	Zipfian
DELETE-1	仅删除	Zipfian
SCAN-1	仅扫描	Zipfian
UPDATE-1	仅更新	Zipfian
WORK-*A*	读取：插入（0.50：0.50）	Zipfian
WORK-*B*	读取：插入（0.95：0.05）	Zipfian
WORK-*C*	读取：删除（0.90：0.10）	Zipfian
WORK-*D*	扫描：插入（0.90：0.10）	Zipfian
INSERT-2	仅插入	Sequence

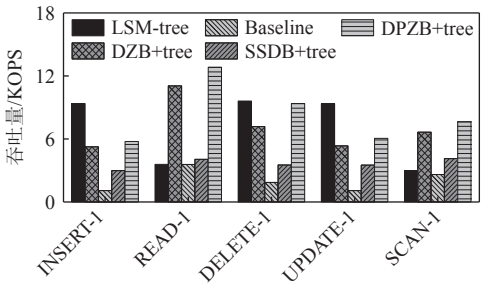
注：括号内的数值为 2 种操作的占比。

本文的实验评估分为 5 个部分：首先，在单一工作负载和混合工作负载下分别测试了各对比对象的吞吐量、尾延迟和对闪存设备的读写次数；其次，测试了各对比对象在写入顺序工作负载下的性能表现；接着，在单一工作负载下评估了不同 PM 使用量对系统性能的影响；随后，测试了各对比对象的正常重启和掉电恢复时间；最后，测试了不同 ZNS SSD 分区大小对 DPZB+tree 的性能影响。

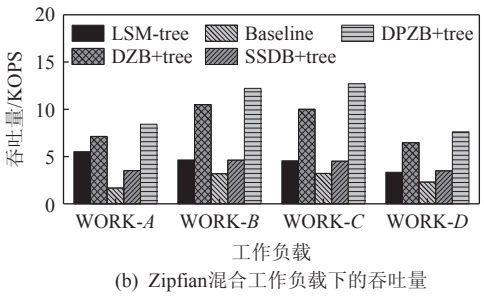
4.3 YCSB 负载测试

1）吞吐量

本节的实验旨在评估在 Zipfian 分布下 DPZB+tree, LSM-tree, DZB+tree, Baseline, SSDB+tree 的吞吐量表现，结果如图 5 所示。对于每组测试都预先载入 300 万条键值对。



(a) Zipfian单一工作负载下的吞吐量



(b) Zipfian混合工作负载下的吞吐量

Fig. 5 Throughput under different workloads

图 5 不同工作负载下的吞吐量

①插入性能。如图 5(a)的插入工作负载所示，DPZB+tree 的插入吞吐量分别是 SSDB+tree 的 1.92 倍、DZB+tree 的 1.1 倍、Baseline 的 5.33 倍。对实验结果进行深入分析可以发现，相较于 Baseline, DPZB+tree 通过多级存储架构并使用 PM 作为存储中间层有效解决了级联更新带来的性能下降问题。而与 DZB+tree 的对比则进一步验证了本文优化策略的有效性，即通过识别写入序列中的热键值来找到当前写入热节点，并将当前写入热节点在写入前迁移到 PM，以此来减少对 ZNS SSD 的写入并将单个写入转化为批量写入。DPZB+tree 的吞吐量低于 LSM-tree，这是因为 B+树的插入操作需要维护叶节点中已排序条目的顺序，当叶节点被填满时，还会触发节点分裂操作，这些操作都会带来额外的开销。而 LSM-tree 在插入数据时，首先将数据顺序追加写入预写日志，随后由后台线程将数据写入内存表。由于 LSM-tree 将条目的

排序以及合并操作推迟到后台的合并阶段进行,因此 LSM-tree 的插入性能要优于 B+树。

②读性能。读操作是索引结构中最基本和最频繁的操作,也是评价一个 B+树类索引性能好坏的重要指标。如图 5(a)所示,DPZB+tree 的读吞吐量分别是 LSM-tree 的 3.58 倍、SSDB+tree 的 3.16 倍、DZB+tree 的 1.16 倍,Baseline 的 3.59 倍。不同于 LSM-tree 的查找方式,DPZB+tree 能在 $O(\log N)$的时间复杂度下查找到键值对,而不是像 LSM-tree 一样如果未命中时需要在内存表中一层层地往下进行查找。因此 DPZB+tree 相较于 LSM-tree 有着较高的查找性能。此外,得益于 DPZB+tree 将内部节点存储在 DRAM 和伪叶节点和热节点数据存储在 PM 的方式,DPZB+tree 在进行查找时,最坏的情况为查找冷叶节点,需要依次读取 DRAM, PM, ZNS SSD,而在查找热节点时仅需读取 DRAM 和 PM。因此 DPZB+tree 有优于其余对比对象的读取吞吐量。

③删除性能。对于 B+树而言,删除操作与插入操作具有同等的重要性,合理的删除策略能够显著地降低系统开销,提升系统性能。DPZB+tree 有与 LSM-tree 相近的吞吐量,并且其吞吐量是 SSDB+tree 的 2.66 倍、DZB+tree 的 1.3 倍、Baseline 的 5.04 倍,这归功于本文使用的删除和合并策略。DPZB+tree 在涉及节点合并时优先与兄弟节点中的热节点合并,这使得该节点可以合并至 PM 以便后续进行高效写入和读取。在此基础上,DPZB+tree 在对冷节点进行删除时会实时查询 PM 剩余容量。若剩余容量能存储 1 个叶节点,则删除对应键值后将该冷节点写入 PM。DPZB+tree 通过将当前删除节点与热节点合并以及迁移策略展现了良好的吞吐量。

④更新性能。对于更新操作来说,各实验对象的总体性能表现与插入操作相当。值得一提的是 DPZB+tree 的更新性能相较于插入性能略有提升,这是因为 INSERT-1 负载虽然是依据 Zipfian 负载生成的,但为了不插入重复的键值,绝大部分操作集中在较小部分的键值上,其并不具备 Zipfian 工作负载的特性。而 UPDATE-1 负载则具有这一特性。因此在测试 UPDATE-1 工作负载时,本文提出的写入请求识别策略能够更为准确地找到当前写入热节点。

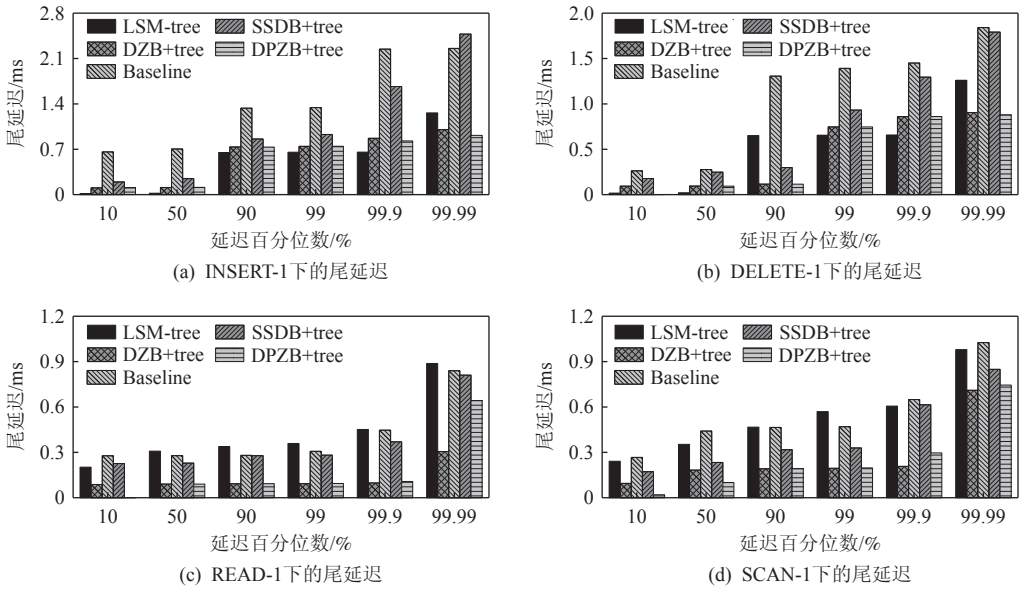
⑤范围查找性能。对于范围查找操作,本文采用给定起始键和查找长度的方式进行测试。DPZB+tree 的吞吐量分别是 LSM-tree 的 2.54 倍、SSDB+tree 的 1.84 倍、DZB+tree 的 1.14 倍、Baseline 的 2.9 倍。对于 DPZB+tree 和 SSDB+tree 来说,跨叶节点的查找操作可以通过叶节点的兄弟指针进行,因此寻找兄弟节点的时间开销较低。而对于 DZB+tree 和 Baseline 来说,在跨叶节点查找时需要根据已查找的路径信息进行回溯。对 LSM-tree 来说,范围查找操作会导致更多的内存表未命中,造成更大的开销。

⑥混合工作负载性能。混合工作负载下各对比对象的性能表现如图 5(b)所示。WORK-*A* 是读写参半的工作负载。在该工作负载下,DPZB+tree 的吞吐量分别是 LSM-tree 的 1.52 倍、SSDB+tree 的 2.40 倍、DZB+tree 的 1.18 倍、Baseline 的 5.03 倍。由于 LSM-tree 主要针对写操作进行优化,在 WORK-*A* 下,DPZB+tree 已经展现出较 LSM-tree 更优异的性能。而 WORK-*B*, WORK-*C*, WORK-*D* 都是读主导的工作负载,在这类工作负载下 DPZB+tree 的性能提升较 LSM-tree, SSDB+tree, DZB+tree 来说更为明显。例如,在 WORK-*C* 工作负载下,DPZB+tree 的吞吐量分别是 LSM-tree 的 2.78 倍、SSDB+tree 的 2.81 倍和 DZB+tree 的 1.27 倍。而对于 Baseline 来说,在 WORK-*C* 下 DPZB+tree 的吞吐量是 Baseline 的 3.94 倍,这比在 WORK-*A* 的工作负载下要低。这是因为 Baseline 的读写性能差距较大,在读主导的工作负载下会获得更好的性能收益,因为写入量减少的缘故,Baseline 无需过多维护从内部节点到叶节点的指针,从而减少级联更新带来的开销。

2)尾延迟

本文测试了各对比对象在 Zipfian 分布下的尾延迟表现。图 6 是分别执行 INSERT-1, DELETE-1, READ-1, SCAN-1 工作负载下的结果。

如图 6(a)所示,在插入操作下,DPZB+tree 在 10%, 50%, 90%, 99%, 99.9% 的尾延迟分布与 DZB+tree 相近,略高于 LSM-tree。而在 99.99% 时,DPZB+tree 展现出最低的尾延迟。LSM-tree 在 99.99% 时,表现出比 DPZB+tree 和 DZB+tree 更高的尾延迟。这是因为 LSM-tree 在内存表写满时,会将其转换为不可变的,并将其刷入磁盘生成一个有序字符串表文件。当某一层级的有序字符串表文件达到预设阈值时,系统就会触发合并操作,这会占用磁盘的 I/O 带宽,从而增加延迟。对于 Baseline 来说,其尾延迟分布呈阶梯状上升,10% 和 50% 相似、90% 和 99% 相似、99.9% 和 99.99% 相似。这归因于 Baseline 的级联更新操作,因此,其插入尾延迟会随着树高的增加而线性增加。此外,SSDB+tree 在 99.99% 时的尾延迟最高,这是由于传统块设备在可用空间低于某个阈值时,可能会不透明地触发垃圾回收操作。



(a) INSERT-1下的尾延迟　(b) DELETE-1下的尾延迟

(c) READ-1下的尾延迟　(d) SCAN-1下的尾延迟

Fig. 6　Tail latency under Zipfian distribution

图 6　Zipfian 分布下的尾延迟

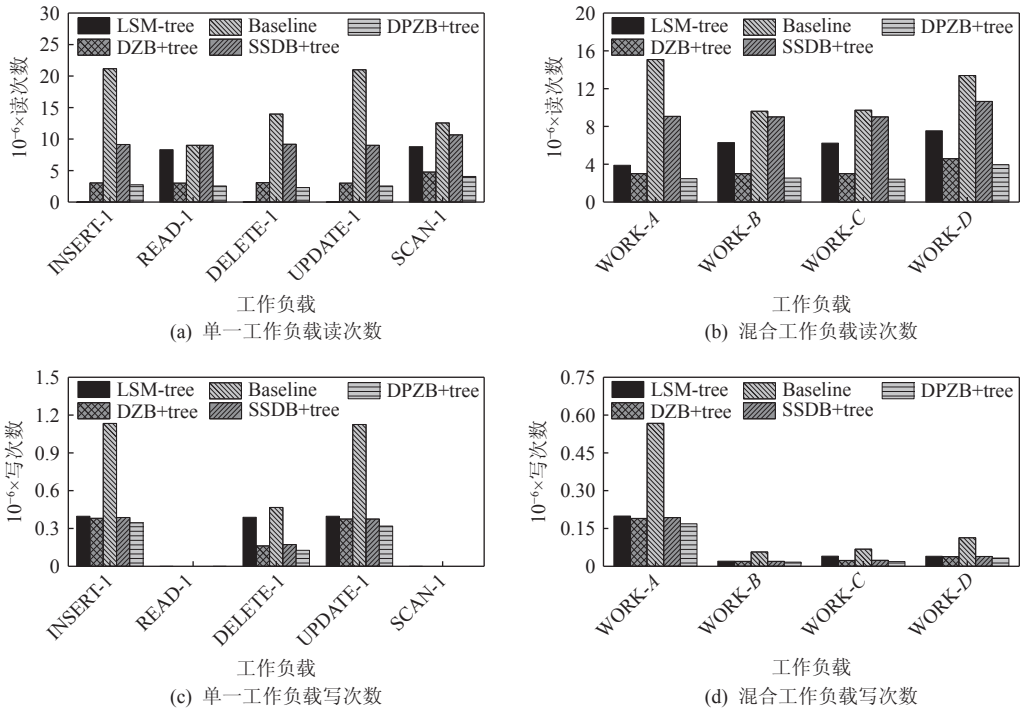
如图 6(b)所示，关于删除操作，DPZB+tree 在 10%, 90%, 99.99% 的尾延迟分布表现出较其他方案更低的尾延迟，这是因为 DPZB+tree 通过写入请求识别机制和节点合并策略有效地增加了后续删除请求在 PM 上的命中率。总体而言 LSM-tree 的删除尾延迟分布与插入尾延迟分布基本一致，这是由于 LSM-tree 的删除特性决定的。LSM-tree 将删除操作转换为写入一个带有删除标志位的键值对，而非立即删除数据，真正的数据删除操作在系统触发合并时执行。除 99% 尾延迟处，DZB+tree, Baseline, SSDB+tree 的删除尾延迟分布较插入尾延迟分布均有不同程度的降低，其中 DZB+tree 降低幅度最为明显，这一结果显而易见。DPZB+tree, DZB+tree, Baseline, SSDB+tree 均属于 B 树类索引结构，其查询耗时随着树高的降低而减少。对于 Baseline 而言，树高的降低还代表需要级联更新的层数减少，从而进一步提升删除操作的性能。

如图 6(c)所示，对于读操作来说，DZB+tree 和 DPZB+tree 在 50%, 90%, 99%, 99.9% 处表现出相近且较其他对比方案较低的尾延迟。而 DZB+tree 在 99.99% 处取得最低的尾延迟，DPZB+tree 在 10% 处取得最低的尾延迟。得益于 DPZB+tree 的写入前热节点识别机制，其在执行操作前将当前写入热键值对应的节点替换到 PM 作为热节点。因此，在当前读取序列中，当读取到与此热键值处于同一叶节点的键值时会直接读取 PM，从而提高读取性能。然而，这一机制的存在也使得 DPZB+tree 在每个读/写序列中可能需要额外执行 1 次写入操作，以确保当前写入热节点已经作为热节点存储在 PM 上。因此，DPZB+tree 在 10% 处的尾延迟表现最优，而在 99.99% 处的尾延迟高于 DZB+tree。对于 LSM-tree 来说，其在 50%, 90%, 99%, 99.9%, 99.99% 处的尾延迟都比其他对比对象高。尽管 LSM-tree 大幅优化了写入性能，但在读取性能方面，LSM-tree 并没有做太多优化。其读取性能的提升主要依赖于合并操作，从而降低在内存表内未命中时从 ZNS SSD 中查询的有序字符串表文件数量。

如图 6(d)所示，对于范围查找操作来说，DPZB+tree 在 90% 和 99% 处的尾延迟与 DZB+tree 表现相近，且较其他对比方案较低。DPZB+tree 在 10% 和 50% 处的尾延迟表现最佳，而 DZB+tree 在 99.9% 和 99.99% 处尾延迟表现最佳。此外，LSM-tree 在 90% 和 99% 处的尾延迟表现最差。Baseline 在 10%, 50%, 99.9%, 99.99% 处的尾延迟表现最差。SSDB+tree 的尾延迟表现总体上优于 LSM-tree 和 Baseline，低于 DPZB+tree 和 DZB+tree。可以发现，较读取操作尾延迟来说，Baseline 在 50%, 99.9%, 99.99% 处的尾延迟都超过了 LSM-tree，这是由于 Baseline 为了降低级联更新的传播，没有设计叶节点的兄弟节点指针。因此，每次范围查找时，Baseline 都需要记录查找路径，以便回溯路径并访问下一个叶节点。

3)读写次数

图 7 展示了在不同工作负载下各对比对象对闪存设备的读写次数。除插入、删除和更新操作外，DPZB+tree 对设备的读次数均低于对比对象。例如，



(a) 单一工作负载读次数 (b) 混合工作负载读次数

(c) 单一工作负载写次数 (d) 混合工作负载写次数

Fig. 7 Read and write frequency under Zipfian distribution

图 7 Zipfian 分布下读写次数

在 READ-1 工作负载下, SSDB+tree 和 Baseline 的读取次数相同, 且是 DPZB+tree 的 3.52 倍。而 LSM-tree 和 DZB+tree 的读次数分别是 DPZB+tree 的 3.25 倍和 1.17 倍。可以看到, LSM-tree, SSDB+tree, Baseline 均表现出较 DPZB+tree 和 DZB+tree 更高的读写次数。SSDB+tree 和 Baseline 的设计决定了其每次查找时都会对闪存设备进行树的高度次的读取。LSM-tree 则是因为在内存表未命中时会对闪存设备进行大量的读取。与之相反, 对于插入、删除和更新操作来说, LSM-tree 将其统一转化为追加写入操作, 而非立即读取或修改键值。只有在有序字符串表合并过程中, 才会涉及对闪存设备的读操作。因此, 对于插入、删除和更新操作来说, LSM-tree 的读次数显著小于对比对象。

对于闪存设备写次数来说, 除 READ-1 工作负载外, DPZB+tree 均表现出较对比对象较低(即在 SCAN-1 负载下写次数都为 0)的写次数。对于 READ-1 工作负载来说, DPZB+tree 会针对当前读取序列进行聚类操作, 进而分析出当前的读热节点, 并在执行读取操作前, 通过热节点处理算法确保当前读热节点已经作为热节点存储在 PM 中。因此, 在将当前读热节点从 ZNS SSD 迁移到 PM 的过程中, DPZB+tree 会产生一定的写次数。

4.4 其他评价指标

1)顺序写入工作负载

4.3 节已经评估了在 Zipfian 分布下 DPZB+tree 和其他对比对象的表现。本节的实验旨在评估在顺序工作负载下(即工作负载的键值分布为递增的, 如写入序列 1, 2, 3, 4, …)各对比对象的性能表现。为简要说明, 本文挑选了在顺序写入下各对比对象的部分性能表现进行展示。同样的, 本实验先载入 300 万条键值顺序增加的键值对, 在存储系统达到稳定状态后, 再插入 300 万条键值顺序增加的键值对, 以对性能进行测试, 最终结果如表 3 所示。

实验结果表明, 在顺序写入工作负载下, DPZB+tree 的性能表现远优于其他对比对象。首先, DPZB+tree 的吞吐量是 LSM-tree 的 3.8 倍、DZB+tree 的 7 倍、Baseline 的 33.8 倍、SSDB+tree 的 14.5 倍。其次, 对于尾延迟来说, DPZB+tree 在 99%, 99.9%, 99.99% 时的尾延迟表现也远远优于对比对象, 具有最低的尾延迟数据。最后, DPZB+tree 相较于对比对象产生了更少的闪存读写次数。

从实验结果中可以看出, DPZB+tree 在顺序写入工作负载下得到了较为显著的性能提升, 这可以归功于本文提出的写入请求识别策略、冷热节点动态放置策略以及叶节点分裂策略。在顺序写入工作负载下, DPZB+tree 能够精准地找到当前写入热节点, 并确保在写入前其已经作为热节点存储在 PM。因此在后续写入时, DPZB+tree 避免了大量的小粒度访

Table 3 Performance of Each Comparative Object in Sequential Write

表 3 顺序写入各对比对象性能表现

性能	LSM-tree	DZB+tree	Baseline	SSDB+tree	DPZB+tree（本文）
吞吐量/OPS	9 531	5 231	1 084	2 515	36 711
延迟在 99% 处/μs	655.8	744.9	1 349.3	1 022.9	16
延迟在 99.9% 处/μs	659.2	875.7	2 247.5	1 847.8	163.5
延迟在 99.99% 处/μs	994.1	1 184.2	2 288.5	2 481.1	684.4
写次数	381 232	383 858	1 133 559	389 834	2 930
读次数	4 359	3 047 318	21 154 061	11 034 439	72

问 ZNS SSD 导致的性能降低。仅在 PM 写满，需要将具有最远访问时间戳的热节点写回到 ZNS SSD 以释放空间给当前写热节点时，才会产生对 ZNS SSD 的写入。所以，相较于对比对象，DPZB+tree 在顺序工作负载下能够取得最佳的性能表现。

2）PM 使用量

本节的实验旨在评估 PM 使用量对系统吞吐量的影响。具体来说，本文选取了 Zipfian 分布下的单一工作负载（INSERT-1, READ-1, DELETE-1, UPDATE-1, SCAN-1），分别测试了在各个操作下，PM 使用量从 ZNS SSD 使用量的 1/50 000（即 2 MB）到 1/5 000（即 20 MB）的性能变化，绘制了吞吐量随 PM 使用量的变化关系图，如图 8 所示。

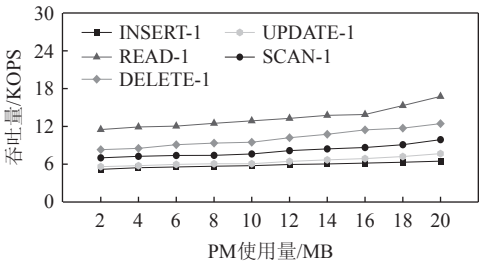


Fig. 8 Throughput changes with PM usage

图 8 吞吐量随 PM 使用量变化情况

对于插入、读取、删除、更新和范围查询操作而言，吞吐量分别实现了 24.8%, 45.8%, 49.7%, 36.8%, 41.2% 的提升。从效率角度来看，插入操作的吞吐量提升最少，每 36% 的 PM 使用量增长仅带来了 1% 的性能收益；而删除操作的吞吐量提升最多，每 18% 的 PM 使用量增长可转化为 1% 的性能收益。

对于各操作而言，PM 使用量的增加均带来了性能的提升。因为 PM 使用量的增加必然会增加在 PM 层中命中读写键值的概率，从而提升效率。同时，从整体来看，PM 使用量的提升对各操作吞吐量的影响较为有限。这一现象表明，即便在 PM 使用量较低的情况下，本文提出的优化策略仍然能够较为显著地提升系统性能，而非单纯依赖于 PM 使用量的增加。

3）恢复时间

如表 4 所示，本文还评估了各对比对象在系统正常重启和掉电恢复的情况下重构索引结构的耗时。对于 B+树索引结构，与正常重启相比，掉电恢复需要扫描叶节点中存储的键值对，以便对掉电时未完成的操作进行恢复。实验结果表明，无论是正常重启还是掉电恢复，DPZB+tree 的耗时都远远低于对比对象。

Table 4 Recovery Time of Each Comparative Object

表 4 各对比对象的恢复时间 s

方案	正常重启	掉电恢复
DPZB+tree（本文）	0.001 2	0.007 7
LSM-tree	0.74	0.76
DZB+tree	281.9	283.51
Baseline	815.38	818.37
SSDB+tree	1.32	1.34

具体而言，DZB+tree, Baseline, SSDB+tree 在恢复过程中均需对所有叶节点进行全盘扫描，以读取最新版本叶节点的最小值，并基于此信息重构树。因此，DZB+tree 将内部节点存储在 DRAM 而叶节点存储在 ZNS SSD 的数据布局方式有利有弊。尽管这种存储架构为 DZB+tree 带来了较高的吞吐量和较低的尾延迟，但同时也显著增加了 DZB+tree 的系统恢复时长。此外，与 DZB+tree 和 Baseline 不同的是，SSDB+tree 是基于 SSD 进行构建的，因此其采用的是就地写入的方式。这种写入方式大大减少了需要扫描的叶节点个数，所以 SSDB+tree 表现出较 DZB+tree 和 Baseline 更短的恢复耗时。

本文提出的 DPZB+tree 在 PM 层存储了所有伪叶节点，并在伪叶节点层保存了对应叶节点的最小键值以及兄弟节点指针。因此，在恢复时 DPZB+tree 只需从 PM 中找到首个伪叶节点，就能通过这一节点访问后续节点，并通过其中保存的叶节点的最小键

值重构整棵 B+树。因为无需访问 ZNS SSD, 所以 DPZB+tree 的恢复耗时远远低于对比对象。

4)分区大小

ZNS SSD 分区大小可由使用者根据应用需求自行决定。目前, ZNS SSD 相关研究设定的分区大小从 128 MB 到 2 GB 不等[45-48]。为评估分区大小对系统性能的影响, 本实验分别测试了不同分区大小下 DPZB+tree 的插入、读取和删除操作吞吐量, 实验结果如图 9 所示。

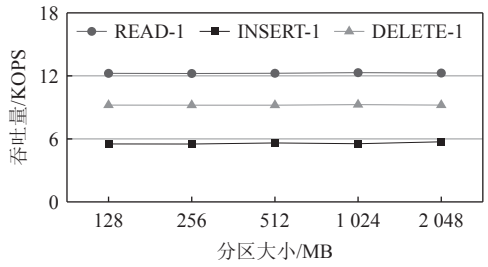


Fig. 9 Throughput changes with zone sizes

图 9 吞吐量随分区大小变化情况

从实验结果可以发现, 系统吞吐量几乎不受分区大小变化影响。这是因为本方案主要聚焦利用持久化内存作为热操作缓冲区, 以降低 ZNS SSD 上的读写放大率, 并不涉及 ZNS 分区内部的数据迁移问题。

5 总结与展望

本文开展面向 ZNS SSD 的 B+树索引结构优化研究, 提出了一种基于 DRAM-PM-ZNS SSD 的冷热数据动态放置的 B+树索引 DPZB+tree。本文对以下 3 个方面进行了研究。首先, 提出写入请求冷热识别策略。通过对当前写入序列的实时分析, 利用低开销 *K*-means 算法进行聚类处理, 从而准确识别并定位当前访问序列的热访问节点。其次, 针对小容量 PM 热数据动态更新以及 PM 容量有限的问题, 提出了基于访问时间戳的冷热节点动态放置策略。最后, 结合各存储介质特性设计了分裂合并策略。实验结果表明, DPZB+tree 在读密集型工作负载和顺序读写工作负载下, 其性能显著优于对比对象。同时, DPZB+tree 还保障了较低的读写延迟和较少的读写次数。通过测试 PM 使用量, 本文还验证了 DPZB+tree 的性能提升并非单纯依靠引入 PM 作为存储中间层。此外, DPZB+tree 还通过 PM 存储伪叶节点, 保存叶节点关键信息, 实现了快速的恢复操作。

未来的工作主要集中在 3 个方向: 1)优化热节点识别机制, 结合深度学习方法提升复杂工作负载下的识别精度; 2)改进冷热节点动态放置策略, 在节点选取时综合考量访问时间戳与存储键值数量, 以降低写放大及后续写开销; 3)完善叶节点的分裂与合并策略设计, 增强其在不同工作场景下的适应性。

作者贡献声明: 曹夕设计了冷热识别策略, 完成了部分实验和撰写论文; 李明杰设计了冷热节点放置策略, 完成了实验代码和撰写论文; 杨朝树设计了分裂合并策略, 修改了部分论文; 杨程搭建了实验环境, 完成了部分实验; 张润宇设计了总体架构, 修改了论文背景和实验部分。

参 考 文 献

[1] International Data Corporation. Revelations in the global DataSphere, 2024: Key trends and takeaways [EB/OL]. (2024-10-18)[2025-01-11]. https://www. idc.com/getdoc.jsp?containerId=prCHC52667624

[2] Yang Chaoshu, Zhuge Qingfeng, Chen Xianzhang, et al. Optimizing synchronization mechanism for block-based file systems using persistent memory[J]. Future Generation Computer Systems, 2020, 111: 288−299

[3] Gao Congming, Shi Liang, Liu Kai, et al. Architecture and technologies of flash memory based solid state drives[J]. Journal of Computer Research and Development, 2021, 58(7): 1518−1532 (in Chinese)
(高聪明, 石亮, 刘凯, 等. 闪存固态硬盘系统结构与技术[J]. 计算机研究与发展, 2021, 58(7): 1518−1532)

[4] Ssdfans. Inside SSD: Solid State Storage Technology, Principle and Practice [M]. Beijing: China Machine Press, 2018 (in Chinese)
(Ssdfans. 深入浅出 SSD: 固态存储核心技术、原理与实战[M]. 北京: 机械工业出版社, 2018)

[5] Zhang Qi, Wang Linzhang, Zhang Tian, et al. Optimized address translation method for flash memory[J]. Journal of Software, 2014, 25(2): 314−325 (in Chinese)
(张琦, 王林章, 张天, 等. 一种优化的闪存地址映射方法[J]. 软件学报, 2014, 25(2): 314−325)

[6] Cui Jinhua, Zhang Youtao, Shi Liang, et al. ApproxFTL: On the performance and lifetime improvement of 3-D NAND flash-based SSDs[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2018, 37(10): 1957−1970

[7] Lu Youyou, Shu Jiwu. Survey on flash-based storage systems[J]. Journal of Computer Research and Development, 2013, 50(1): 49−59 (in Chinese)
(陆游游, 舒继武. 闪存存储系统综述[J]. 计算机研究与发展, 2013, 50(1): 49−59)

[8] Bjørling M. From open-channel SSDs to zoned namespaces [C/OL]//Proc of the Linux Storage and Filesystems Conf (Vault'19). Berkeley, CA: USENIX Association, 2019[2025-01-11]. https://www.

usenix.org/conference/vault19/ presentation/bjorling

[9] Maheshwari U. From blocks to rocks: A natural extension of zoned namespaces [C] //Proc of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'21). New York: ACM, 2021: 21−27

[10] Stavrinos T, Berger D S, Katz-Bassett E, et al. Don't be a blockhead: Zoned namespaces make work on conventional SSDs obsolete [C]// Proc of the Workshop on Hot Topics in Operating Systems (HotOS' 21). New York: ACM, 2021: 144−151

[11] Samsung. Samsung introduces its first ZNS SSD with maximized user capacity and enhanced lifespan [EB/OL]. (2021-06-02)[2025-01-11]. https://semiconductor.samsung.com/cn/newsroom/news/samsung-introduces-its-first-zns-ssd-withmaximized-user-capacity-and-enhanced-lifespan/

[12] Western Digital. Zoned storage SSDs [EB/OL]. (2022-12-05)[2025-01-11]. https://www.westerndigital.com/solutions/zns-ssd

[13] Zhang Weihua, Zhao Chuanlei, Peng Lu, et al. High performance GPU concurrent B+tree [C] //Proc of the 27th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming (PPoPP'22). New York: ACM, 2022: 443−444

[14] Black J P, Taylor D J, Morgan D E. A robust B-tree implementation [C] //Proc of the 5th Int Conf on Software Engineering (ICSE'81). Los Alamitos, CA: IEEE Computer Society, 1981: 63–70

[15] Oracle Corporation. MySQL 8.0.37 reference manual: InnoDB [EB/OL]. (2024-04-30) [2025-01-11]. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html

[16] Microsoft Corporation. NTFS technical overview [EB/OL]. (2024-02-11) [2025-01-11]. https://learn.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview

[17] Sweeney A, Doucette D, Hu Wei, et al. Scalability in the XFS file system [C] //Proc of the USENIX Annual Technical Conf (ATC'96). Berkeley, CA: USENIX Association, 1996: 1−14

[18] He Gongshan, Zhao Chuanlei, Jiang Jinhu, et al. A survey of data storage technologies for deep learning[J]. Chinese Journal of Computers, 2025, 48(5): 1013−1064 (in Chinese)
(贺巩山, 赵传磊, 蒋金虎, 等. 面向深度学习的数据存储技术综述[J]. 计算机学报, 2025, 48(5): 1013−1064)

[19] Han Shukai, Xiong Ziwei, Jiang Dejun, et al. Rethinking index design based on persistent memory device[J]. Journal of Computer Research and Development, 2021, 58(2): 356−370 (in Chinese)
(韩书楷, 熊子威, 蒋德钧, 等. 基于持久化内存的索引设计重新思考与优化[J]. 计算机研究与发展, 2021, 58(2): 356−370)

[20] Xiao Renzhi, Feng Dan, Hu Yuchong, et al. A survey of data consistency research for non-volatile memory[J]. Journal of Computer Research and Development, 2020, 57(1): 85−101 (in Chinese)
(肖仁智, 冯丹, 胡燏翀, 等. 面向非易失内存的数据一致性研究综述[J]. 计算机研究与发展, 2020, 57(1): 85−101)

[21] Boukhobza J, Rubini S, Chen Renhai, et al. Emerging NVM: A survey on architectural integration and research challenges[J]. ACM Transactions on Design Automation of Electronic Systems, 2018, 23(2): 1−32

[22] Yang Jian, Kim J, Hoseinzadeh M, et al. An empirical guide to the behavior and use of scalable persistent memory [C] //Proc of the 18th USENIX Conf on File and Storage Technologies (FAST'20). Berkeley, CA: USENIX Association, 2020: 169−182

[23] Kim S H, Shim J, Lee E, et al. Empowering storage systems research with NVMeVirt: A comprehensive NVMe device emulator[J]. ACM Transactions on Storage, 2023, 19(4): 1−26

[24] Bjørling M, Aghayev A, Holmberg H, et al. ZNS: Avoiding the block interface tax for flash-based SSDs [C] //Proc of the USENIX Annual Technical Conf (ATC'21). Berkeley, CA: USENIX Association, 2021: 689−703

[25] Western Digital. Linux zoned storage support overview [EB/OL]. (2022-11-15)[2025-01-11]. https://zonedstorage.io/docs/introduction

[26] Western Digital. Ultrastar DC ZN540 ZNS SSD product brief [EB/OL]. (2020-12-30)[2025-01-11]. https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/product-brief/product -brief-ultrastardc-zn540.pdf

[27] NVM Express. NVMe specifications overview [EB/OL]. (2021-06-03)[2025-01-11]. https://nvmexpress.org/specifications/

[28] Western Digital. Linux zoned storage support overview [EB/OL]. (2023-11-22)[2025-01-11]. https://zonedstorage.io/docs/linux

[29] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33(4): 351−385

[30] Siying D, Kryczka A, Jin Yanqin, et al. Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience [C] //Proc of the 19th USENIX Conf on File and Storage Technologies (FAST'21). Berkeley, CA: USENIX Association, 2021: 33−49

[31] Lee H R, Lee C G, Lee S, et al. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs [C] //Proc of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22). New York: ACM, 2022: 93−99

[32] Wu Denghui, Liu Biyong, Zhao Wei, et al. ZNSKV: Reducing data migration in LSMT-based KV stores on ZNS SSDs [C] //Proc of the 40th IEEE Int Conf on Computer Design (ICCD'22). Piscataway, NJ: IEEE, 2022: 411−414

[33] Tan Zhenhua, Long Linbo, Shen Jingcheng, et al. Optimizing garbage collection for ZNS SSDs via in-storage data migration and address remapping[J]. ACM Transactions on Architecture and Code Optimization, 2024, 21(4): 1−25

[34] Liang Yupei, Chen Tseng Yi, Chi Ching Ho, et al. Enabling a B^+-tree-based data management scheme for key-value store over SMR-based SSHD [C] //Proc of the 57th ACM/IEEE Design Automation Conf (DAC'20). Piscataway, NJ: IEEE, 2020: 1−6

[35] Do J, Picoli I L, Lomet D, et al. Better database cost/performance via batched I/O on programmable SSD[J]. VLDB Journal, 2021, 30(3): 403−424

[36] Liu Yang, Jin Peiquan. ZB+-tree: A novel ZNS SSD-aware index structure[J]. Journal of Computer Research and Development, 2023, 60(3): 509−524 (in Chinese)
(刘扬, 金培权. ZB+-tree: 一种 ZNS SSD 感知的新型索引结构[J]. 计算机研究与发展, 2023, 60(3): 509−524)

[37] Duffy C, Shim J, Kim S H, et al. Dotori: A key-value SSD based KV store[J]. Proceedings of the VLDB Endowment, 2023, 16(6): 1560−

1572

[38] Zhang Hongbin, Fan Jie, Shu Jiwu, et al. Summary of storage system and technology based on phase change memory[J]. Journal of Computer Research and Development, 2014, 51(8): 1647−1662 (in Chinese)
(张鸿斌, 范捷, 舒继武, 等. 基于相变存储器的存储系统与技术综述[J]. 计算机研究与发展, 2014, 51(8): 1647−1662)

[39] Li Wen, Wang Ying, Liu Cheng, et al. On-line fault protection for ReRAM-based neural networks[J]. IEEE Transactions on Computers (TC), 2023, 72(2): 423−437

[40] Yang Jianguo, Luo Qing, Xue Xiaoyong, et al. A 9 Mb HZO-based embedded FeRAM with 10^{12}-cycle endurance and 5/7ns read/write using ECC-assisted data refresh and offset-canceled sense amplifier [C] //Proc of the IEEE Int Solid-State Circuits Conf (ISSCC'23). Piscataway, NJ: IEEE, 2023: 498−499

[41] Manohar S S, Kapoor H K. CAPMIG: Coherence-aware block placement and migration in multiretention STT-RAM caches[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2023, 42(2): 411−422

[42] Cooper B F, Silberstein A, Tam E, et al. Benchmarking cloud serving systems with YCSB [C] //Proc of the 1st ACM Symp on Cloud Computing (SoCC'10). New York: ACM, 2010: 143−154

[43] Western Digital. Getting started with emulated zoned block devices [EB/OL]. (2021-11-22)[2025-08-10]. https://zonedstorage.io/docs/getting-started/zbd-emulation

[44] Li Huaicheng, Hao Mingzhe, Tong M H, et al. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator[C]// Proc of the 16th USENIX Conf on File and Storage Technologies (FAST'18). Berkeley, CA: USENIX Association, 2018: 83−90

[45] Long Linbo, He Shuiyong, Shen Jingcheng, et al. WA-Zone: Wear-aware zone management optimization for LSM-Tree on ZNS SSDs[J]. ACM Transactions on Architecture and Code Optimization, 2024, 21(1): 1−23

[46] Liu Renping, Chen Junhua, Chen Peng, et al. Hi-ZNS: High space efficiency and zero-copy LSM-Tree based stores on ZNS SSDs[C]// Proc of the 53rd Int Conf on Parallel Processing. New York: ACM, 2024: 1217−1226

[47] Wang Yu, Zhou You, Lu Zhonghai, et al. Flexzns: Building high-performance ZNS SSDs with size-flexible and parity-protected zones[C] //Proc of the 41st IEEE Int Conf on Computer Design (ICCD'23). Piscataway, NJ: IEEE, 2023: 291−299

[48] Zhang Runyu, Zhou Lening, Li Mingjie, et al. Efficient wear-leveling-aware data placement for LSM-Tree based key-value store on ZNS SSDs[J]. Journal of King Saud University-Computer and Information Sciences, 2024, 36(7): 102156



Cao Xi, born in 2001. Master candidate. Student member of CCF. Her main research interests include non-volatile memory and storage systems.
曹　夕, 2001 年生。硕士研究生。CCF 学生会员。主要研究方向为非易失性内存、存储系统。



Li Mingjie, born in 2000. Master candidate. His main research interests include non-volatile memory and storage systems.
李明杰, 2000 年生。硕士研究生。主要研究方向为非易失性内存、存储系统。



Yang Chaoshu, born in 1984. PhD, associate professor, master supervisor. Member of CCF. His main research interests include non-volatile memory, high-performance computing, and storage systems.
杨朝树, 1984 年生。博士，副教授，硕士生导师。CCF 会员。主要研究方向为非易失性内存、高性能计算、存储系统。



Yang Cheng, born in 1999. Master candidate. His main research interest includes storage systems.
杨　程, 1999 年生。硕士研究生。主要研究方向为存储系统。



Zhang Runyu, born in 1995. PhD, associate professor, master supervisor. Member of CCF. His main research interests include non-volatile memory, high-performance computing, and storage systems.
张润宇, 1995 年生。博士，副教授，硕士生导师。CCF 会员。主要研究方向为非易失性内存、高性能计算、存储系统。