计算机科学
COMPUTER SCIENCE

基于持久内存的B+树索引优化综述

卢超, 杨朝树, 姚政竹, 刘颖, 张润宇

引用本文

卢超, 杨朝树, 姚政竹, 刘颖, 张润宇. 基于持久内存的B+树索引优化综述[J]. 计算机科学, 2026, 53(1): 77-88.

LU Chao, YANG Chaoshu, YAO Zhengzhu, LIU Ying, ZHANG Runyu. Survey on Optimization B+ Tree Index for Persistent Memory [J]. Computer Science, 2026, 53(1): 77-88.

相似文章推荐（请使用火狐或 IE 浏览器查看文章）

Similar articles recommended (Please use Firefox or IE to view the article)

基于Inverted-B+树的海量三维地质块体模型高效索引方法

Efficient Indexing Method for Massive 3D Geological Block Models Based on Inverted-B+ Tree

计算机科学, 2025, 52(8): 146-153. https://doi.org/10.11896/jsjkx.240700127

基于大语言模型自身的提示语公平性自动优化与评估

Automatic Optimization and Evaluation of Prompt Fairness Based on Large Language Model Itself

计算机科学, 2025, 52(4): 240-248. https://doi.org/10.11896/jsjkx.240900008

面向NewSQL数据库数据协同持久化的研究

Study on Collaborative Data Persistence in NewSQL Databases

计算机科学, 2025, 52(1): 131-141. https://doi.org/10.11896/jsjkx.231200079

基于B+树存储的AABB包围盒碰撞检测算法

Collision Detection Algorithm of AABB Bounding Box Based on B+ Tree

计算机科学, 2021, 48(6A): 331-333. https://doi.org/10.11896/jsjkx.200600113

用数据驱动的编程模型并行多重网格应用

Parallelizing Multigrid Application Using Data-driven Programming Model

计算机科学, 2020, 47(8): 32-40. https://doi.org/10.11896/jsjkx.200500093

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.250200109

基于持久内存的B+树索引优化综述

卢 超 杨朝树 姚政竹 刘 颖 张润宇
贵州大学计算机科学与技术学院 贵阳 550025
省部共建公共大数据国家重点实验室 贵阳 550025
(gs.clu23@gzu.edu.cn)

摘 要 持久内存的出现为索引结构设计提供了新思路,同时在数据一致性、持久化开销和并发控制等方面也带来了设计挑战。作为存储系统中应用广泛的索引结构,B+树亟需针对持久内存的硬件特性进行适配优化,以充分发挥其字节寻址、非易失性和低延迟等优势。围绕持久内存上B+树索引优化问题,首先分析了构建基于持久内存B+树所存在的挑战,其次分别从单一持久内存架构和混合内存架构两个视角综述了优化方案。对于单一持久内存架构,总结了数据一致性方案、并发控制优化和叶节点创新设计的研究进展,探讨了如何在保证瞬时恢复的基础上提升写操作效率;对于DRAM-PM混合架构,分析了基于叶节点结构优化和基于辅助结构优化的策略,总结了如何在选择性持久化的基础上提升索引性能。最后,总结并分析了两类架构下不同方案的设计特点及优缺点,并对未来在两类架构下的B+树索引优化发展方向进行了展望。

关键词:持久内存;B+树;读写优化;数据一致性;持久化开销
中图分类号 TP311

Survey on Optimization B+ Tree Index for Persistent Memory

LU Chao,YANG Chaoshu,YAO Zhengzhu,LIU Ying and ZHANG Runyu
College of Computer Science and Technology,Guizhou University,Guiyang 550025,China
State Key Laboratory of Public Big Data,Guiyang 550025,China

Abstract The advent of persistent memory(PM) introduces new perspectives for index structure design while presenting challenges in data consistency,persistence overhead,and concurrency control. As a widely adopted index structure in storage systems,the B+ tree requires tailored adaptations to harness PM's unique features,including byte-addressability,non-volatility,and low latency. This paper focuses on the optimization of B+ tree indexes for persistent memory,beginning with an analysis of the challenges in designing PM-based B+ trees. Then,optimization strategies are reviewed from two perspectives:PM-only architectures and DRAM-PM hybrid architectures. For PM-only architectures,advancements in data consistency mechanisms,concurrency control optimizations,and innovative leaf node designs are summarized,with an emphasis on enhancing write operation efficiency while ensuring instant recovery. For DRAM-PM hybrid architectures,strategies based on leaf node structure optimization and auxiliary structure enhancement are examined,highlighting approaches to improve indexing performance through selective persistence. Finally,a detailed analysis of the design characteristics,advantages,and limitations of optimization schemes under both architectures is presented,offering insights into future research directions for B+ tree index optimization in these contexts.

Keywords Persistent memory,B+ Tree,Read-write optimization,Data consistency,Persistence overhead

1 引言

随着云计算、大数据和人工智能等技术的迅猛发展,数据处理和存储的需求急剧增加,传统存储解决方案面临巨大的挑战。当前,许多商业数据库系统依赖机械硬盘(HDD)和固态硬盘(SSD)作为存储介质。尽管这些设备能够提供较大的存储容量,但其高访问延迟和较低的吞吐量往往无法满足现代应用对性能的苛刻要求。动态随机存取存储器(DRAM)虽然在速度上具有优势,但其易失性、高功耗和较高的存储成本[1],限制了其大规模应用的可扩展性。

持久内存(Persistent Memory,PM)凭借其非易失、可字节寻址、低延迟、低功耗和大容量等特点[2-6],为计算机和存储系统的发展带来了新的机遇和挑战。随着持久内存技术不断发展,多种新型存储方案相继出现,包括相变存储器(Phase

到稿日期:2025-02-26 返修日期:2025-06-20
基金项目:国家自然科学基金地区科学基金(62162011,62362009);贵州大学引进人才科研项目([2022]44)
This work was supported by the Fund for Less Developed Regions of the National Natural Science Foundation of China(62162011,62362009) and Talent Introduction Project of Guizhou University([2022]44).
通信作者:张润宇(zhangry@gzu.edu.cn)

Change Memory)[7-10]、自旋转移力矩存储器(Spin-Transfer Torque RAM)[11-13]、电阻式存储器(Resistive RAM)[14-15]、铁电存储器(Ferroelectric RAM)[16-17]和 3D XPoint[18-20]等。英特尔于 2019 年发布了基于 3D XPoint 技术的傲腾持久内存(Optane DC Persistent Memory),这是第一款针对主流计算机系统的商用持久内存。

持久内存技术的发展填补了主存和现有持久存储技术之间的空白,从根本上改变了存储系统的设计方式。作为存储系统的核心组件,B+树也需要重新设计,以充分发挥持久内存的硬件特性。如果仅将传统针对块设备设计的 B+树应用于持久内存,则将无法有效利用其性能优势。因此,如何设计面向持久内存优化的 B+树成为亟待解决的问题。

近年来,随着持久内存技术的发展,研究人员针对 B+树的优化方案逐渐分化为两大方向:单一持久内存(PM-only)优化[21-32]和混合内存(DRAM-PM)优化[33-42],如图 1 所示。

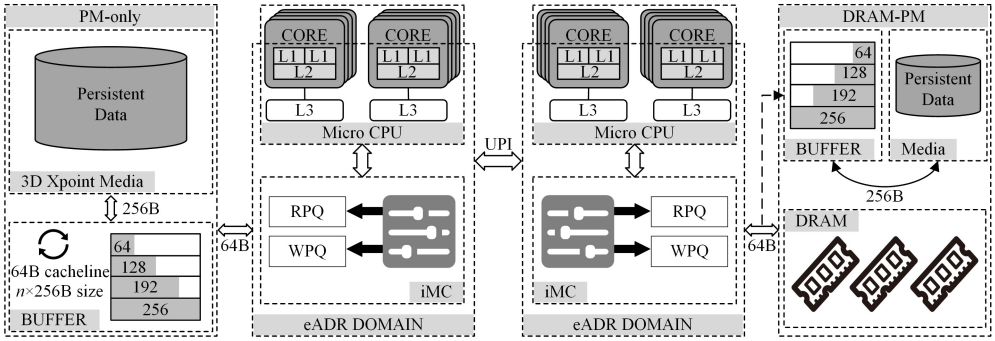

图 1 持久内存系统架构

Fig. 1 Architecture of persistent memory system

本文旨在总结近年来基于持久内存的 B+树索引优化方案。第 1 章介绍了新型存储技术的相关背景;第 2 章梳理了构建基于持久内存的 B+树所面临的挑战;第 3 章和第 4 章分别从单一持久内存和混合持久内存两种架构出发,系统总结并归纳了相关方案;第 5 章则对持久内存中 B+树的设计方案进行了总结和展望,并讨论了未来持久内存上 B+树的优化方向;最后总结全文。

2 构建基于持久内存 B+树的挑战

在传统存储系统中,B+树作为一种高效的索引结构,因其优异的范围查找能力和稳定的查询性能,被广泛应用于数据库[43-44]、文件系统[45-49]及其他键值存储系统中[50-56]。其作为核心索引组件,对提升存储系统性能具有关键作用。在数据库系统中,B+树凭借其高扇出度,在较少的层级中容纳更多数据,从而降低树的高度,减少磁盘 I/O 次数,并提升数据处理速度。由于数据库需频繁进行数据插入,B+树在节点分裂和平衡维护方面的高效性对保持系统性能至关重要。在文件系统中,B+树的层级结构便于组织和管理文件元数据,从而快速定位文件和目录,显著减少 I/O 访问次数。此外,在基于混合存储介质的 LSMTree 键值存储系统中,B+树因其就地更新能力和优秀的范围查找性能,常被用作辅助索引层,以加速系统的性能。在 HDD 环境中,B+树的实现通常结合磁盘块的访问特性,将节点大小设计为磁盘块大小的倍数,以优化读取效率;而在 SSD 环境中,B+树的平衡特性有助于减少写放大效应,集中写入更新节点,从而降低数据分布不均所带来的性能开销。由于持久内存具有可字节寻址、低延迟、非易失性、读写不对称和大容量等特点,使得基于持久内存构建的 B+树面临数据一致性、持久化开销和并发控制挑战。

2.1 数据一致性

现代计算机系统为了解决 CPU 处理速度和内存访问速度严重不匹配的问题,在 CPU 和 DRAM 之间引入高速缓存(Cache),Cache 在系统断电后数据会全部丢失。尽管持久内存具备非易失性,但是在数据实际写入到持久内存时,即数据到达异步内存刷新(Asynchronous DRAM Refresh, ADR)域[57]之前,可能会暂时驻留在 CPU 缓存中,并随时以任意顺序透明地写回内存控制器写挂起队列。若系统发生断电或崩溃,缓存中尚未写入持久内存的数据将丢失,可能导致 B+树在持久内存上的状态不一致。因此,为保证数据一致性,必须通过适当的机制将数据及时写回持久内存,以避免因断电或系统故障引发的数据丢失问题。虽然 ADR 机制能够保证系统断电或崩溃后,位于 ADR 域内的数据都会被持久化,但是目前 CPU 缓存并不位于 ADR 域内[57-59]。为了有效保障数据的持久性,程序必须显式地调用缓存行刷新指令(如 CLFLUSH),才能将 CPU 缓存中的脏数据强制写回到 ADR 域中,以确保数据在持久内存的可靠写入。

此外,持久内存提供的失败原子写粒度与传统块设备不

单一持久内存上的研究充分利用持久内存的非易失性,使得位于持久内存的整个 B+树能实现瞬时故障恢复,这些研究还提出了多种创新设计。wB+Tree[21]提出通过排序数组来维持叶节点无序条目上的读性能。Fast&Fair[22]则利用 B+树无重复指针特性实现可容忍的瞬时不一致性。此外,CircTree[31]的循环叶节点设计减少了插入时的条目移动开销。其他方案如 BzTree[25]和 PACTree[26]在无锁并发和日志异步修改方面取得进展。在混合内存研究中,研究者结合 DRAM 以寻求性能与持久性之间的平衡。NVTree[33]提出的内部节点重建方案通过仅保障叶节点的一致性来优化性能。FPTree[34]也引入选择性持久化方案,通过将内部节点存放于 DRAM 中来提升访问速度。LB+Tree[35]探索了缓存行内字修改次数对 3D XPoint 性能的影响,并提出了新的优化方向。CCL-BTree[42]提出叶节点缓冲策略,将缓冲节点放置于 DRAM 中,以有效减少傲腾内存上的写放大现象。

同。具体来说,机械硬盘的原子写粒度为扇区,固态硬盘的原子写粒度为闪存页[29,37,60],而持久内存仅支持 8 字节的原子写粒度[61]。原子写粒度与缓存行刷新粒度的差异为数据一致性保障带来了的新挑战。当更新超过 8 字节数据时,需要由相应机制实现一致性更新,以防止系统崩溃后出现数据丢失或部分更新。现有的持久内存数据一致性方案主要采用写时拷贝或预写日志,通过在更新数据之前将副本写入其他地方供恢复使用。以写时拷贝方案为例,Condit 等[62]提出了短路写时拷贝,通过利用持久内存的 8 字节原子写粒度,直接在父节点中就地更新节点指针,避免了写时拷贝过程中叶节点指针修改引发的级联更新,从而减少了严重的写放大。但由于 B+树结构内存在兄弟节点指针,无法通过短路写时拷贝同时处理多节点更新的一致性。

2.2　持久化开销

现代处理器通过改变内存操作顺序来充分利用内存带宽,内存控制器会对写入进行重排序。这种机制对基于持久内存的 B+树提出了严格的写入顺序要求,使得设计高效写入的 B+树更具挑战。例如,当对位于 PM 中的叶节点执行插入操作时,新插入的键值对必须被完全写入 PM 之后,才能更新叶节点的元数据,否则更新元数据后叶节点将包含无效键值对,造成数据不一致。为了不影响持久内存 B+树的数据一致性,使用缓存行刷新(CLFLUSH 等)和内存屏障(MFENCE)指令来保证有序的内存写入。缓存行刷新指令能够将待写入数据刷新到持久内存中,另一方面,内存屏障指令能够确保在屏障之前的内存操作先于屏障之后的内存操作完成,两者结合形成有序的内存写入。

同时,由于传统存储设备的读写延迟远高于 DRAM,基于传统存储设备的 B+树索引性能主要受限于 I/O 路径中的高延迟,而内存层面的读写开销相对可以忽略。相比之下,持久内存的读写延迟接近 DRAM,远低于传统存储设备,这使得基于持久内存的 B+树性能高度依赖于持久化指令的开销。频繁调用持久化刷新指令会显著增加性能损耗,降低 B+树在持久内存上的写入效率,对 B+树的设计提出了更高的要求。因此,在设计时需尽量降低持久化操作的频率,从而减小持久化操作带来的性能开销。

2.3　并发控制

多核心架构 CPU 已成为现代计算机的主流,为了充分发挥其性能,高效的并发控制成为提升索引性能的关键。在 B+树中,插入和删除操作可能引发节点分裂或合并,涉及多个节点的修改。并发环境下,多个线程可能同时访问或修改相同节点,进而引发数据竞争、脏读和更新丢失等问题。持久内存的非易失特性使得这些问题在系统崩溃后仍然存在,迫使传统并发控制增加额外步骤以确保崩溃后能正确恢复。因此,除了确保在崩溃恢复的一致性(避免锁状态丢失、数据损坏或死锁等),并发控制机制还须具备良好的可扩展性。

3　基于单一持久内存的 B+树研究进展

在单一持久内存架构下构建的 B+树索引,凭借持久内存的非易失特性,能够在系统崩溃或断电等故障发生后实现瞬时恢复。因此,设计能够正确恢复的 B+树索引,以提高系统的可用性和稳定性,成为一个重要的研究问题。为此,研究人员结合持久内存特性与 B+树索引特点,提出了多种数据一致性保障方案。同时,为了提升写操作效率和并发处理能力,研究者还针对并发控制机制进行优化,并探索了单一持久内存下的叶节点设计。本章从 3 个方面对单一持久内存架构下的 B+树研究进行梳理:数据一致性方案、并发控制机制优化以及叶节点创新设计。表 1 总结了基于单一持久内存架构的 B+树优化方案。

表 1　单一持久内存上 B+树优化方案

Table 1　PM-only B+Tree optimization schemes

索引	实验环境	一致性机制	并发性	叶节点组织
wB+Tree[21]	PM 模拟器	基于位图原子写和只读日志	无	无序,维护排序数组
FAST&FAIR[22]	PM 模拟器	利用总存储有序(TSO)的存储依赖性来原子写入	无锁查找	有序
PB+Tree[23]	Optane DCPMM	节点版本计数器	乐观无锁搜索和读写锁	有序
SSBTree[24]	Optane DCPMM	Lazy-Box 日志机制	乐观并发控制和细粒度写锁	有序
BzTree[25]	NVDIMM	持久性多字比较并交换(PMwCAS)	无锁	无序
PACTree[26]	Optane DCPMM	基于位图原子写和 SMO 日志	乐观持久化版本锁	无序
WOPETree[27]	Optane DCPMM	原子更新元数据	细粒度乐观版本锁	缓冲区无序,键值对数组有序
WOTree[28]	PM 模拟器	叶节点 WAL 机制	无锁查找和读锁	无序
IsleTree[29]	PM 模拟器	原子更新元数据	节点锁数组	缓存行内有序,行间无序
TLBTree[30]	Optane DCPMM	持久内存原子写	无	无序
CircTree[31]	PM 模拟器	写入要求严格的修改顺序	细粒度锁	有序并且视为循环圆
CCTree[32]	PM 模拟器	原子更新元数据	无锁查找	无序

3.1　数据一致性方案

数据库系统中广泛研究的写时拷贝或预写日志方案应用到基于持久内存的 B+树中开销较大。数据更新前的副本写入以及持久化指令的调用延迟,会削弱持久内存带来的性能优势。下述方案结合了 PM 和 B+树索引结构的特点,设计出了更加轻量化的一致性方案。

2015 年 Chen 等[21]提出的 wB+Tree,为了减小传统的撤销重做日志(Undo-redo Logging)的开销,引入了 NewRedo 和 CommitNewRedo 方案。在写入未使用过地址时,NewRedo 只记录地址和新值,CommitNewRedo 延迟刷新日志,并执行一次 mfence,有效减少了日志写开销。更极端情况下,当写入的值在提交前不会再次被访问时,WriteRedoOnly 仅记录写意图,延迟实际的写入操作,在统一刷新日志后执行写入,减小了每次写操作的开销。针对叶节点层的新建节点操作,短路写时拷贝虽然能一致更新父节点指针,但仍需更新叶节点兄弟指针,短路写时拷贝无法处理多指针原子更新。

wB+Tree 利用 WriteRedoOnly 优化方案延迟更新父节点和兄弟节点指针，解决了短路写时拷贝无法处理多个指针更新的一致性，保证了新建叶节点的一致性。同时，wB+Tree 通过 8 字节原子写操作更新表示键值条目有效性的位图(Bitmap)，以保障数据一致性，如图 2 所示。插入操作通过位图快速定位空闲条目并完成写入，无需移动，通过更新位图实现删除条目。然而，wB+Tree 未考虑节点分裂带来的写开销和并发操作冲突，在分裂节点时依赖重做日志保证崩溃一致性，从而降低了写性能。

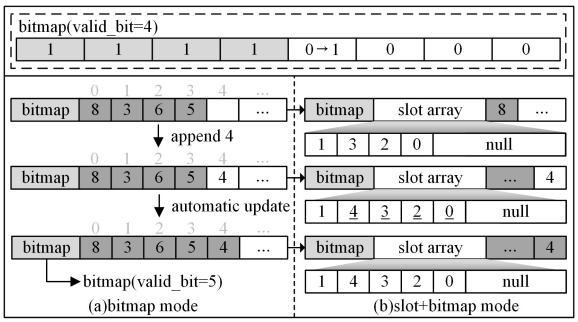

图 2 位图模式与位图结合插槽模式

Fig. 2 Bitmap mode and bitmap with slot mode

2018 年 Hwang 等[22]提出了 FAST&FAIR，其中叶子节点层采用 B+树结构，内部节点层采用 B-link 树结构，以提高并发性并保证崩溃一致性。其提出的故障原子转移(FAST)和故障原子就地重平衡(FAIR)算法，利用 8 字节原子写指令使得 B+树能够转换为其他一致状态，或保持瞬时不一致状态以容忍读操作，从而避免昂贵的日志记录和读锁开销，并使读事务不被阻塞。X86 架构强制总体存储指令有序[63-64]，当存储和加载指令之间存在依赖关系，内存访问操作的顺序不会任意更改。基于这一观察，FAST 算法在执行插入操作时，将大于插入键的条目从后往前移动，无需为每个条目的移动调用缓存行刷新和内存屏障指令来保证排序要求；删除操作则以插入操作的相反顺序执行。然而，FAST 算法要求脏缓存行必须按顺序刷新。如果在移动过程中发生崩溃，则根据 B+树中不存在相同指针的性质识别数据不一致。FAIR 算法在分裂节点或合并节点时采用就地更新策略。在分裂节点过程中，完成一半节点的迁移后，设置原节点的中间指针为空，从而使新节点有效。使用 FAST 算法插入新条目并更新父节点，如果发生崩溃，重启后可以识别并恢复不一致的数据。然而，维护节点条目的有序性会在插入、删除等操作中引发大量的写开销。FAST 和 FAIR 算法保证读操作不会访问到不一致的节点状态，读操作按节点内数据移动的方向进行线性搜索，确保读取正确的数据，避免阻塞并发读操作。Hwang 等[22]的实验表明，插入过程中 wB+Tree 调用缓存行刷新指令次数是 FAST&FAIR 的 1.7 倍，FAST&FAIR 的范围查询比 wB+Tree 快 33%。

2022 年 Yoo 等[23]指出 FAST&FAIR 存在插入性能随节点规模扩大而下降、FAST 算法增加内存访问延迟，以及无锁搜索偶发结果错误 3 种局限性，为此提出 PB+Tree，引入了 pivot key 和 split key 两个额外的元数据。其中 split key 用于纠正 FAST&FAIR 中并发插入问题，在 FAST&FAIR 中没有将中间的 key 存储在正确的兄弟节点中，导致了并发插入的正确性问题。而 pivot key 则用于在创建右兄弟节点时，将被迁移的键值对分为两个子数组存放在不同区域，在分裂时减少键值对移动。PB+Tree 以动态偏移量替代 FAST&FAIR 的哨兵指针，同时采用乐观无锁搜索算法，避免重复访问同一树节点，虽然乐观无锁算法涉及访问不正确的子节点，但 PB+Tree 使用双兄弟指针去保证搜索结果的正确性。在 TPCC 基准测试中 PB+Tree 的查询处理吞吐量是 FAST&FAIR 的 3.7～11.5 倍。

2022 年 Li 等[24]提出了 SSBTree，一种结合 Lazy-Box 机制和优化乐观并发控制策略的 B+树。SSBTree 旨在实现高效的崩溃一致性和瞬时恢复能力，尤其适用于高可用存储系统。它在节点内部引入 Lazy-Box 日志技术，通过聚集连续的节点修改操作并选择性地使用写时拷贝(Copy-on-Write，COW)技术，以 8 字节原子粒度提交节点修改操作(Node Modification Operation，NMO)，从而降低写放大效应并确保数据一致性，无需额外的恢复算法。在 COW 优化方面，SSBTree 采用左右并列(Side-to-Side)技术，通过重用节点空间减小写时拷贝的开销。具体来说，节点空间被分为左右两部分，写时拷贝时仅需将已用部分拷贝至未用部分，从而显著减小开销。SSBTree 还引入了放松的树结构设计，允许节点延迟更新父节点，将树的 SMO 操作转化为 NMO 操作，从而简化了结构维护。利用 Lazy-Box 的失败原子性执行特性，无需额外恢复机制来实现瞬时恢复。由于读操作需要维护父节点的延迟更新，SSBTree 在乐观并发控制基础上对读操作进行优化，避免了因等待锁而阻塞。同时，写操作在保证崩溃一致性的前提下，也能够高效执行。Li 等[24]的实验表明，相比 FAST&FAIR，SSBTree 在插入和删除上分别提高了 29%和 40%的吞吐量，但占用的 PM 空间是 FAST&FAIR 的 2 倍，并且在系统崩溃后不需要额外恢复机制。

3.2 并发控制机制优化

常见的 B+树并发控制方法包括锁耦合(Lock Coupling)、乐观锁(Optimistic Locking)和无锁技术(Lock-free)。由于 PM 的数据持久性，需要对这些方法进行调整以适应持久性的要求。例如，锁的状态也需要持久化，以避免在崩溃后锁的状态丢失，导致死锁或数据损坏。因此，并发控制不仅要处理多线程的竞争，还要确保崩溃一致性。

2018 年 Arulraj 等[25]提出了 BzTree，通过使用持久性多字比较并交换(PMwCAS[65])原语实现了无锁 B+树。传统的无锁索引实现通常使用 CAS 硬件原语，其仅支持原子修改单字。对于 B+树的合并和分裂过程，需要进行多字修改，但使用多个 CAS 操作会使得索引存在多个中间状态，增加无锁并发算法的设计难度并降低性能。PMwCAS 允许在无锁情况下原子地更改多个任意 8 字节持久内存单元，且保证崩溃一致性，不会出现中间状态，只有初始状态或修改完成状态，大幅降低了并发实现的复杂度。内部节点以键的顺序存储记录，以实现快速查找，除了对子指针的更新外，内部节点采用写时拷贝进行更新。叶节点包含空闲空间，在插入和更新时进行缓冲，并会定期将空闲空间中的无序条目合并到已排序

的存储区，后续插入可继续写入空闲空间。由于 BzTree 并不维护叶节点的兄弟指针，其范围查找性能较差。

2021 年 Kim 等[26]提出了用于设计高性能持久索引结构的打包异步并发指南，并基于此提出了 PACTree。PACTree 基于并发自适应基数树[66](ART)提出持久可线性化基数树(PDL-ART)作为内部节点层，并发控制采用读优化写互斥[67](ROWEX)。叶子节点层采用双向链表，每个叶子节点包含锚键指示最小键值，并有 64 个键值对。为优化读操作，指纹数组和间接排序数组大小对齐缓存行，间接排序数组内容不持久化，而是根据需要从一致的键值对中重新生成，以减少 PM 写入。通过 8 字节原子写操作更新位图，原子地提交对叶节点中条目的更改。为避免结构化修改操作(Structural Modification Operation，SMO)阻塞内部节点的并发访问并成为并发扩展性瓶颈，PACTree 提出了基于日志的异步结构修改方法。叶节点分裂时，拆分操作先记录到 SMO 日志中并立即提交，不继续修改父节点。后台线程根据 SMO 日志在搜索层中添加新数据节点或删除过时数据节点。虽然异步结构修改将开销较大的 SMO 操作与关键路径解耦，但可能导致搜索层和数据层状态不一致，查询无法通过搜索层找到新分裂的节点。此外，PACTree 使用非统一内存访问感知(Non-uniform Memory Access，NUMA-aware)的分配器，从本地 NUMA 节点分配持久内存，以避免跨 NUMA 节点的持久内存访问带来的性能下降。Kim 等[26]的实验表明，PACTree 比 BzTree 的插入延迟更短以及范围查询更快。

2024 年 He 等[27]提出了 WOPETree，旨在优化 IOT 设备低功耗场景下 B+树的存储效率与并发性能。首先，WOPETree 采用缓存行对齐的刷新写缓冲(Flush-aligned Write Buffer，FWB)策略(见图 3)，将插入操作聚集至单个缓存行刷新粒度中，以低开销实现条目的持久化更新。该策略确保新条目和元数据能够通过单次缓存行刷新指令提交，显著降低写入成本。其次，现有的面向 PM 的 B+树在节点分裂时需迁移原节点一半的条目，导致大量写开销。因此 WOPETree 在节点分裂过程中采用选择性迁移策略(Selective Migration of Node Entries，SMNE)，根据 FWB 中的数据分布情况预测未来写入趋势，并细粒度决定迁移条目数量。这有效避免了在写密集场景下，叶节点频繁分裂加剧写放大的问题。最后，WOPETree 设计了一种细粒度乐观版本锁(Fine-grained Optimistic Version Lock，FOVL)并发控制方案，提高了读写线程对同一节点的并行访问效率。FOVL 机制通过将写操作限制在叶节点中受影响的条目上，仅锁定写入条目，避免阻塞未修改区域的查询操作。读写线程可以无锁地在同一节点上并行操作，写线程将插入条目记录到 FWB 中，然后原子地更新元数据。这样可以避免影响键值数组上的并发读取，解决了读写线程之间的数据竞争问题，减小了写锁对查询性能的影响。然而，由于 FWB 较小，写操作密集时，仍需频繁地将数据从缓冲区刷新到键值对数组中，会产生较大的写开销。在多线程场景下，WOPE 的写吞吐量略高于 SSBTree 和 FAST&FAIR，由于同一节点上可同时读写，WOPE 的读吞吐量相比 SSBTree 和 FAST&FAIR 提升较大。

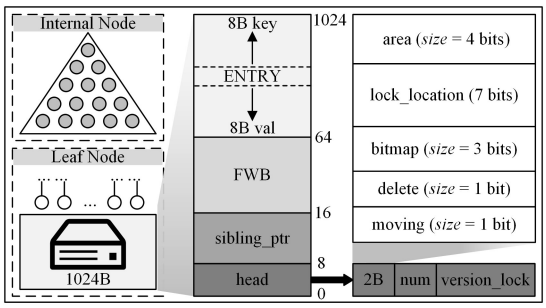

图 3　WOPE 树架构

Fig. 3　Architecture of WOPE Tree

3.3　叶节点创新设计

传统 B+树要求节点内部的键值对有序存储，插入或删除操作平均需要移动一半条目。对基于持久内存的 B+树来说，维护节点内键值条目有序将带来大量的 PM 写开销。Chen 等[68]提出叶节点采用无序方式组织，可以避免插入或删除时频繁的数据条目移动，较大地减少了 PM 写入次数。

wB+Tree[21]采用无序节点设计，然而查找时需要线性扫描。为弥补叶节点无序的缺点，wB+Tree 引入了间接排序数组，用于维护键值对的逻辑顺序，每个槽位存储对应条目的偏移量，支持二分查找以提高查找效率。然而，维护该间接索引数组也会带来额外的 PM 写操作。Ma 等[28]提出了 WOTree，其叶节点直接采用 wB+Tree 的位图和槽数组方案，在保留 wB+Tree 优点的基础上进一步优化写。同时 WOTree 提出延迟持久化策略，仅在分裂操作完成后对迁移的节点执行缓存行刷新，而非逐一对迁移的键值对执行刷新操作。在数据一致性方面，由于内部节点可通过扫描叶节点进行重建和修改，WOTree 仅对叶节点实施 WAL 机制，以最小化一致性开销。与 wB+Tree，FAST&FAIR 相比，WOTree 插入操作引起的缓存行刷新次数分别减少了 22.2%和 30.8%，执行时间分别缩短了 27.3%和 44.7%。

2020 年 Wang 等[29]提出了 IsleTree，采用将叶子节点划分为多个与缓存行对齐的空间，每个缓存行内条目有序，但节点内缓存行之间的条目无序。与 FAST&FAIR[22]要求叶节点内所有条目有序不同，IsleTree 的设计使得大部分插入或删除操作仅会引发一个缓存行的刷新开销，而不需要移动大量条目，避免了多个缓存行的刷新。在叶节点中，条目按半排序方式组织，查找操作使用跳跃式线性搜索。当目标条目的键值小于当前搜索条目的键值时，直接跳跃到下一个缓存行中继续搜索。此外，不同于传统 B+树的节点分裂操作，IsleTree 将要分裂的节点数据一分为二，将一半条目迁移到两个新节点中，并将两个新节点插入叶子节点层，同时更新父节点。然而，IsleTree 的节点分裂机制仍然会导致大量的数据拷贝和叶子节点层的修改，从而产生较高的写开销。

2021 年 Luo 等[30]提出 TLBTree。由于持久内存具有较大写开销的特点[69-71]，现有的持久化 B+树研究大多集中在减小写开销以提升写性能，但往往牺牲了读性能。因此，很少有方案能够同时提供良好的读写性能。基于 B+树的搜索层频繁读取而叶节点层频繁写入这一观察，提出了两层持久索

引(Two-Layer Persistent Index, TLPI)架构。该架构解耦了搜索层和叶节点层,允许不同的索引设计分别优化读写性能。搜索层采用数组组织优化读性能,叶节点层通过叶节点数组和大量子树优化写性能,数组用于定位子树。然而,双层数组结构不适合插入和删除操作,会导致大量数据迁移。此外,下层大量独立子树的设计也会导致范围查询性能的显著下降。TLBTree主要聚焦于其两层架构下的读写性能优化,但节点删除、分裂等操作仍然存在较大的写开销。

2022年Wang等[31]提出CircTree。研究发现,采用叶节点内部条目有序结构的持久内存B+树(如FAST& FAIR[22])在插入或删除键值对时,会引发大量键值对单向移动,造成严重的写放大问题。同时,迁移键值对需要按序刷新脏缓存行,会带来较大的写开销。针对大量键值对单向移动的问题,CircTree提出了逻辑循环圆结构的树节点设计(见图4)。该设计中,节点仍以数组方式组织有序键值对条目,但在逻辑上视其为无固定基地址的循环圆结构。插入或删除操作时,键值对可以在节点内双向移动。根据键值对在循环结构中的相对位置,选择合适的方向进行较少数量的键值对移动,从而显著减少条目迁移数量。最理想的情况下,向节点插入最小或最大键值时,无需移动其他条目,逻辑上仅需在两端追加键值。节点的元数据存储条目的起始基地址和数量,用于确定循环结构的左右边界。循环节点中的条目双向移动依赖于取模运算,而传统的取模运算通常依赖开销较大的整数除法。CircTree将条目数量设计为2的幂,取模运算被优化为按位与操作。尽管该设计有效减少了条目迁移数量并优化了取模运算,但是节点内条目排序的开销依然存在。

2024年Yan等[32]为了解决在维护PM上的数据一致性时会遭受严重的性能下降问题,提出了CCTree,该树遵循如下原则:通过持久化原语确保叶节点数据持久化,最小化对PM和DRAM的指针解引用,以及减少对同一缓存行的重复写入。其内部节点保持有序以支持快速搜索,而叶节点无序以优化写入性能。元数据被拆分为头部(含锁、版本号)和尾部(含位图、兄弟指针),分别置于节点两端以避免缓存行争用。内部节点头部元数据包含min_key和max_key,其中min_key支持快速定位子节点,max_key旨在降低兄弟节点访问频率;不同于内部节点将min_key与max_key放在一起,叶节点则通过将元数据拆分为位于头部的元数据0和位于尾部的元数据1,min_key和max_key则分别放在元数据0和元数据1处,其利用位图快速定位空闲槽位,并通过3种锁状态(未锁定、锁定和动态修改锁定)控制并发访问。若节点分裂导致max_key不一致,则触发二次搜索兄弟节点。对比PACTree,FAST&FAIR和ROART,在搜索、插入、更新、删除、范围查找和混合负载场景下,分别有1.2～1.6倍、1.5～1.7倍、1.5～2.8倍、1.9～4倍、0.9～10倍和1.55～1.82倍的性能提升。

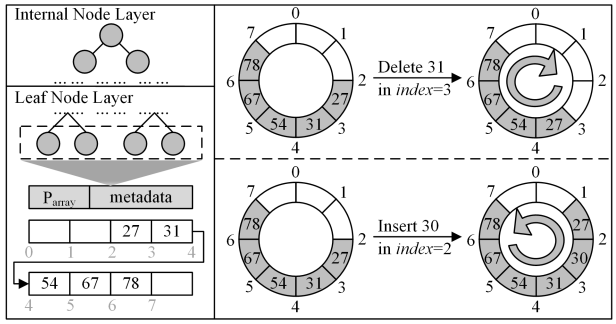

图4 Circ树架构图

Fig. 4 Architecture of CircTree

4 基于DRAM/PM混合内存的B+树研究进展

多数基于持久内存的B+树方案将整个树结构存储在PM上,虽然实现了系统崩溃后的瞬时恢复,但就访问速度而言,比直接位于DRAM上的索引方案慢,在性能上存在显著挑战。为了应对这一挑战,Oukid等[34]首次提出了在混合架构(即DRAM结合PM)下构建B+树,将内部节点放置于DRAM中来优化索引遍历速度。该方案尽管牺牲了部分瞬时恢复能力,但使得索引访问速度显著提高。研究人员在此基础上提出了不同的性能优化方案。根据写优化策略的不同,混合架构下的B+树优化方法可归纳为以下两个方面:基于叶节点结构的优化和基于辅助结构的优化。本章将对上述两类优化方案进行分析与比较。表2总结了基于DRAM-PM混合内存架构的B+树优化方案。

表2 DRAM-PM混合内存架构上B+树优化方案

Table 2 DRAM-PM hybrid memory B+ tree optimization schemes

索引	实验环境	一致性机制	并发性	叶节点组织
NVTree[33]	NVDIMM	追加原子写入	无	无序,追加写
FPTree[34]	SCM模拟器	基于位图原子写和微持久日志	硬件事务内存	无序,维护指纹数组
LB+Tree[35]	Optane DCPMM	持久内存原子写	硬件事务内存	无序,维护指纹数组
CrabTree[36]	PM模拟器	写时复制和就地移位	无	有序
uTree[37]	Optane DCPMM	原子修改链表指针	细粒度版本锁	链表层节点有序
ROWETree[38]	Optane DCPMM	自验证原子写入	细粒度版本锁	半无序,缓存行内有序
HWTree[39]	Optane DCPMM	原子更新元数据	版本控制锁	无序
DPTree[40]	Optane DCPMM	写优化自适应日志和原子写	并行合并树组件	无序,维护排序数组
HBTree[41]	Optane DCPMM	持久化日志和原子写	无	有序
CCL-BTree[42]	Optane DCPMM	保守写日志和原子写	细粒度版本锁	无序,追加写

4.1 基于叶节点结构优化

这类方案根据B+树的结构特点进行优化。由于B+树的内部节点仅用于索引,实际数据都存储于叶节点层,写操作都主要集中在叶节点层。考虑到PM相比DRAM具有较高的写延迟,研究人员充分利用叶节点位于PM中的特点,提出了多种优化叶节点结构的方案来提升混合架构下的B+树性能。

2015年Yang等[33]提出NVTree。由于PM相比DRAM具有更高的读写延迟,基于单层PM架构设计的B+树[21,33,72]性能较差。最先提出的CDDSTree[72]在维护叶节点有序条目时需多个缓存行刷新,并确保内部节点一致性,导致

整体一致性开销较大。NVTree 首次提出了选择性保证数据一致策略，仅保证叶节点数据一致性，内部节点通过一致的叶节点层重建，并存储在 DRAM 中，显著降低了一致性维护开销。为了避免插入时移动条目的刷新开销，NVTree 保持叶节点条目无序存储，并通过在键值对中加入标识来区分插入和删除操作，将其转化为日志结构的追加写（见图 5）。叶节点条目无序并采用追加写，导致读操作每次检索所有条目，因此性能较差。NVTree 通过 8 字节原子更新叶节点中的键值对数量字段，使写入条目可见，无需使用版本控制或日志记录来恢复崩溃状态。NVTree 还优化了内部节点的组织格式，所有内部节点存储在连续的内存空间中，提高了空间利用率和读取性能。然而，内部节点无法拆分，任意节点分裂必须重建整个内部节点层，导致在多线程环境下成为性能瓶颈。

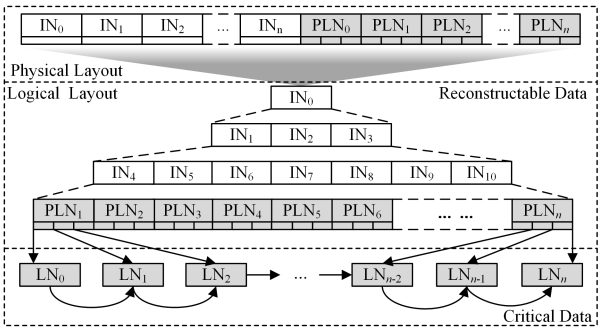

图 5　NVTree 架构图

Fig. 5　Architecture of NVTree

2016 年 Oukid 等[34]提出了基于 DRAM-PM 混合存储架构的 FPTree。在该设计中，叶节点存储在 PM 中，内部节点存储在 DRAM 中，并且在恢复时重建（见图 6）。因此，树遍历过程中，只有访问叶节点时的开销比完全易失性的 B+树更大。FPTree 使用硬件事务内存（Hardware Transactional Memory，HTM）并发控制内部节点，以提升性能。由于 HTM 与缓存行刷新指令冲突，位于 PM 上的叶节点的并发控制采用细粒度锁。叶节点内的键值对无序存储，并增加位图表示插槽有效性。为减小线性搜索无序条目开销，FPTree 提出了指纹技术加速搜索。每个键值计算 1 字节哈希值，存储在叶节点开头，检索条目时首先扫描指纹，从而将探测次数限制为一次。然而，FPTree 在分裂时需要日志记录来保证一致性，导致其分裂开销较大。Oukid 等[34]用 FPTree 替换掉 Memcached（分布式内存键值存储）的哈希表索引，多线程下吞吐量仅损失了大约 2%，与完全位于 DRAM 中的哈希表性能几乎一样，且并发扩展性好。

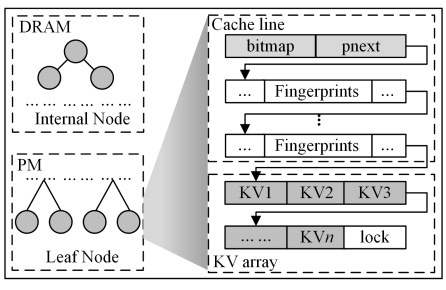

图 6　FPTree 架构图

Fig. 6　Architecture of FPTree

2020 年 Liu 等[35]提出了针对 3DXPoint 内存的 LB+Tree。虽然 3DXPoint 内存符合之前对非易失性内存的假设，但也具有一些独特特征。例如，缓存行内修改字的数量不会影响写入性能，内部数据访问粒度为 256 字节。基于这些特征进一步提出了条目移动方案，如图 7 所示。新条目插入到元数据所在的第一个缓存行空插槽中，一次缓存行写入即可插入新条目和更新元数据。若第一个缓存行已满，则需两次写入：插入新条目到另一个空缓存行并迁移第一个行中的条目，为第一个缓存行创建新的空插槽，最大程度地减少缓存行刷新指令的调用次数。LB+Tree 延续 FPTree[34]的选择性持久化方案，并使用 HTM 进行并发控制。在元数据中使用 FPTree 的指纹技术[34]加速查询。为降低依赖日志记录保证节点分裂一致性所带来的额外写开销，LB+Tree 利用 8 字节原子写操作保证分裂操作的一致性。Liu 等[35]的实验探讨了在 3DXpoint 容量足够大的情况下，将 X-Engine 的跳表替换为 LB+Tree 和选择性持久化跳表，实验证明 LB+Tree 在单线程和多线程下均比跳表性能更优。

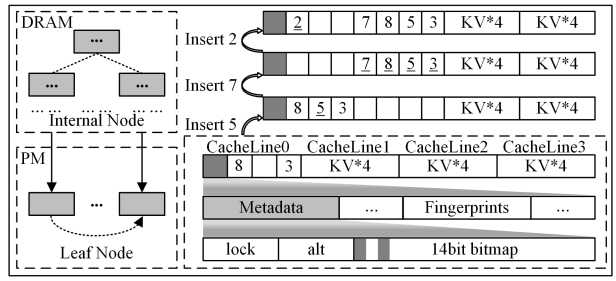

图 7　LB+树架构

Fig. 7　Architecture of LB+Tree

2020 年 Wang 等[36]基于 ARMv8 架构对持久内存的支持，在此架构下提出了 CrabTree。该树采用了 NVTree 的选择性数据一致性保证方案，仅对叶节点层实施崩溃一致性保障。针对叶节点插入或删除操作中大量键值对移动所带来的持久化开销问题，CrabTree 提出了两种策略：写时拷贝和批量数据转移（Shifting in Place，SIP）。在每次插入或删除操作中，CrabTree 根据一致性成本选择开销最小的策略。具体而言，COW 将所有 KV 对复制到新叶节点中，并在批量刷新后替换原节点；SIP 通过有序刷新缓存行来移动叶节点中的 KV 对。但动态写入策略会引入计算开销，同时该设计在其他架构下会存在局限性。对比 FAST&FAIR 和 NVTree，CrabTree 在写和读性能方面提高了 2.2 倍和 3.7 倍。

2020 年 Chen 等[37]提出了 uTree，旨在解决 B+树在 PM 中存在的高尾延迟问题。研究指出，高尾延迟的主要原因有两个：1）内部结构修改操作频繁触发的延迟；2）在高并发情况下，PM 写入带宽限制，导致并发操作之间发生严重干扰。uTree 在 DRAM 中构建完整的 B+树，但叶节点不再存储键值对，而是存储键和持久指针（称为数组层叶节点），通过持久指针引入位于 PM 中的影子链表层，每个节点仅包含一个条目，并通过指针链接形成单链表。这一设计有效避免了 PM 中叶节点排序、分裂和合并带来的写入开销，尽管增加了频繁的内存分配开销。链表层基于元素粒度组织，减少了多个线程操作时因叶节点条目共享同一锁而引发的阻塞问题。

uTree通过一系列比较并交换(Compare and Swap,CAS)指令,在链表层实现了细粒度的无锁并发控制。此外,由于DRAM具有较低的写入延迟,索引层的B+树仍采用现有的粗粒度锁并发方案,显著提升了系统的并发性能并降低了延迟。Chen等[37]的实验表明,写吞吐量在2.2Mop/s的情况下,uTree的第99.9百分位尾延迟仅为5 μs,比FPTree低一个数量级。

2022年Zou等[38]提出了ROWETree,其是一种针对DRAM-PM混合架构设计的读优化、写高效的B+树结构。现有持久内存优化B+树索引难以同时实现高效的读写性能,主要原因在于查询优化与插入性能之间存在权衡。无序叶节点设计虽能提升写入性能,但降低了搜索性能。即便引入额外的元数据(如排序数组[21]或指纹[34])来减小扫描开销,却以牺牲写入性能为代价来提高读取性能。ROWETree提出了自验证插入和半排序叶节点技术,实现了读写性能的良好平衡。叶节点采用追加写入方式,并利用8字节键作为持久化标记,避免了额外元数据维护,且每次插入仅需一个缓存行刷新指令。叶节点内同一缓存行中的键值条目保持有序,搜索时可跳过不必要的比较,提升读性能的同时不影响写入性能。由缓存行内排序引起的额外写入不会损害写入性能,因为缓存行内修改字的数量不会影响写入性能[35]。为缓解PM中的读写干扰,ROWETree在DRAM中构建热数据缓存,将热数据的访问转移至DRAM,并通过轻量级机制周期性跟踪热数据变化,以确保同步。Zou等[38]对现有方案在叶节点插入条目时,所需的持久化指令调用次数进行了总结。因为元数据的更新,NVTree需要刷新2次。由于位图和指纹数组的维护,FPTree则需要刷新3次,而LB+Tree由于条目迁移操作需要刷新2次。与此不同,ROWETree采用自验证式追加写入,每次插入仅需一次缓存行刷新,显著降低了持久化开销。

2024年Yan等[39]针对PM场景下B+树的写性能低效、恢复延迟高等问题,提出了HWTree。针对无序键值对布局的叶节点,HWTree提出了分散式验证方案,将位图更新的开销均摊到每个键值对中,从而缓解了插入或删除时更新位图导致的连续写问题。其核心在于给每个值(Value)都附加1位有效位(有效置为1,无效置为0),通过单缓存行刷新操作完成写入,消除了集中更新元数据导致的PM连续写问题。同时,HWTree通过节点两端的影子位图实现故障安全的分裂验证机制,两个位图记录了移位过程之后的有效键值对信息。更新时先更新右位图,然后更新左位图,若二者不一样,则意味着失败,利用右位图来进行重建。在崩溃恢复环节,其引入重启点标记机制,每次根节点更新时自动记录各子树最左边的叶节点地址作为恢复入口,崩溃后由多线程沿兄弟指针遍历分区重建,将大规模键值对的恢复时间压缩至分钟级。但叶节点内键(Key)仍无序存放,导致查询时最坏情况要遍历叶节点所有数据。对比FPTree,DPTree,uTree和PACTree,HWTree在插入、搜索和范围查找方面的性能分别提高了1.5～2.73倍、0.97～4.72倍和2.6～13倍,且HWTree的重建恢复时间缩短了88%。

4.2 基于辅助结构优化

叶节点采用位图作为元数据的方案,在写入一个KV后,首先需要确保KV被刷入PM中,然后更新位图。通常原子更新位图需要一次单独的缓存行刷新。为了降低混合架构下B+树的持久化开销,研究人员引入辅助索引结构实现批量写入,将元数据的持久化开销进行均摊,并通过辅助索引结构来提升读性能。

2019年Zhou等[40]提出了DPTree。现有持久化索引[21-22,33-34,72]在维护结构属性或更新元数据时需要执行大量持久化原语。为降低持久化原语开销,DPTree提出以批量写入的方式更新索引,将一批键值条目插入到PM中的叶节点时,仅需更新一次元数据,从而将持久化开销均摊。DPTree基于两级索引架构[1]设计,第一层索引称为缓冲树,由位于DRAM中的B+树构成,并通过PM中的写优化重做日志来保证持久性。第二层索引称为基础树,只读并采用选择性持久化方案,DRAM中使用基数树作为索引,PM中由叶子节点链表构成。查找时首先搜索缓冲树,若未找到则搜索基础树;范围查询则在所有树中搜索并合并结果。插入、删除和更新操作写入缓冲树中,当缓冲树的大小超过阈值时,会将其合并到基础树中。在并发控制方面,缓冲树使用乐观锁耦合[67](Optimistic Lock Coupling,OLC)支持并发操作。当缓冲树达到阈值时,会转化为中间缓冲树,并创建新的缓冲树来替代当前缓冲树,后台线程负责合并基础树和中间缓冲树。合并过程较为昂贵,可能导致前台线程被阻塞,从而降低索引的插入性能。Zhou等[40]的实验表明,DPTree的插入延迟比FAST&FAIR和FPTree分别低1.27倍和1.76倍。DPTree相比FAST&FAIR具有更好的线程扩展性。

2021年Zhou等[41]为了提高现有持久内存索引的运行效率并缩短恢复时间,基于DRAM-PM架构提出了HBTree。整个索引结构被划分为数据层、中间层和索引层。数据层采取将全局B+树划分为多棵日志树的组织方式,每个日志树由PM树、Cache树和日志机制组成。PM树是位于PM上的小规模B+树,按连续键范围组织数据,从而减少全局B+树高度增加带来的频繁PM访问,降低性能损耗。HBTree进一步统计PM树中各键的访问频率,识别热数据并将其缓存至DRAM上的Cache树。通过PM上的日志机制,HBTree保证了Cache树的一致性并定期与PM树同步。在读密集场景下,大部分访问通过DRAM处理,显著提升了读性能。中间层管理数据层中的所有日志树,采用双向链表结构组织。每个链表节点代表一棵日志树,存储热数据的统计信息,以便识别热点数据并加速读写操作。为缩短恢复时间,中间层节点存储在持久内存中。索引层由DRAM中的B+树组成,用于索引中间层节点的元数据。在崩溃发生后,可通过遍历中间层快速重建索引层。实验表明,HBTree的性能随着热数据比例增加而提高。

持久内存与DRAM一样集成在内存总线上,以64字节缓存行粒度处理请求。当数据写入PM时,CPU缓存行的内容首先写入PM上的写合并缓冲区(XPBuffer),然后转换为256字节的数据单元(XPLine)写入物理介质[69-71]。2024年Li等[42]发现,缓存行和写合并缓冲区导致在随机写入小键值

对时引发写放大问题,具体表现为,由缓存行引起的 CLI 放大和由 XPLine 引起的 XBI 放大。尽管现有研究多聚焦于减少缓存行刷新次数以缓解 CLI 放大,但 XBI 放大对 PM 写性能影响更为显著。DPTree 利用位于 DRAM 中的全局缓冲池来缓冲与合并写请求,未能充分缓解 XBI 放大问题。为此,CCL-BTree 提出了一种基于叶节点的缓冲策略,在最后一层内部节点与叶节点之间引入缓冲节点,利用 DRAM 缓冲区减少 XPLine 刷新次数,从而缓解 XBI 放大问题(见图 8)。叶节点存储在 PM 中,而内部节点和缓冲节点则保留在 DRAM 中,以合并写请求并降低刷新频率。为了确保崩溃一致性,CCL-BTree 设计了写保守日志方案。在将新键值对插入缓冲节点的空插槽之前,首先在 PM 中的预写日志(Write-Ahead Logging,WAL)上追加日志条目。当触发缓冲节点刷新时,省略额外的日志记录,减少不必要的日志开销,从而缓解预写日志引发的 XBI 放大。为避免日志垃圾回收(Garbage Collection,GC)过程中对 PM 产生随机写入,CCL-BTree 提出了局部感知垃圾回收技术。该技术在写入缓冲节点前,将键值对记录到主日志中,并在 GC 过程中利用辅助日志存储待写入 PM 的键值对。回收完成后,辅助日志成为新的主日志。这些策略有效减少了 XBI 放大的影响,显著提升了写入吞吐量。

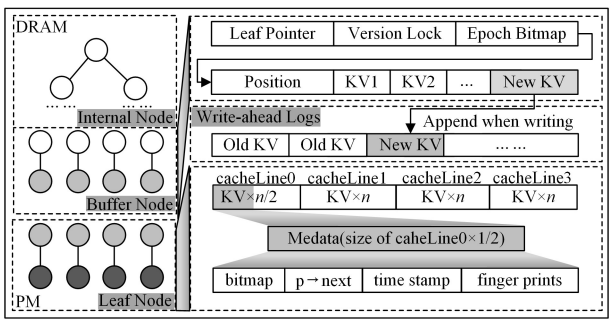

图 8　CCL-BTree 架构图

Fig. 8　Architecture of CCL-BTree

5　总结与展望

本章分别对单一持久内存和混合内存架构下 B+树设计和优化进行总结,并对基于持久内存的 B+树未来的发展方向进行展望。

在单一持久内存架构中,整颗 B+树存储于 PM 中。为了充分利用持久内存的非易失性,现有研究主要聚焦于以最低的一致性维护代价,实现系统崩溃或断电后的索引瞬时恢复,确保索引的高可用性。同时,针对持久内存读写性能不对称(读优于写)的特性,PM-only 架构通过保证叶节点的有序性来维持高效的范围查找性能。在此基础上,提出了多种叶节点组织方式,或适度放宽 B+树结构约束,以减小写操作的性能开销并提升写性能。此外,考虑到持久内存相比传统块存储具有显著的低延迟优势,现有研究还提出了多种并发控制方案,以在多核架构下充分发挥持久内存的优势,提升索引的并发效率并减小由锁操作带来的额外开销。

在混合内存架构中,由于当前持久内存的读写延迟高于 DRAM,且带宽低于 DRAM,单一持久内存上的 B+树性能远不及完全位于 DRAM 中的 B+树。为了提升性能,选择性持久化方案通过将 B+树的内部节点存储于 DRAM 中,来实现接近 DRAM 完全存储的 B+树性能,从而达到性能与持久性之间的平衡。同时,针对 8 字节原子写与 64 字节缓存行刷新粒度不匹配的问题,现有研究提出了将元数据与插入键值对置于同一缓存行的叶节点组织方式。该方案通过单次缓存行刷新实现插入操作与元数据更新的同步,在保证一致性的同时显著降低持久化开销。此外,考虑到持久内存写延迟和带宽的局限,现有研究还提出了批量写入方案。系统在持久内存上采用优化的预写日志机制,来保证数据一致性。同时,将写入请求在 DRAM 中进行批量聚集与顺序化处理,在触发阈值后统一刷新到持久内存。该方案不仅提升了读性能,还有效减小了写入过程中的持久化开销。

基于持久内存的 B+树优化已取得快速发展,但仍面临诸多亟待解决的问题。本文展望了未来可能的研究方向。

1)随着大数据时代的到来,传统 B+树在处理 PB 级甚至 ZB 级数据时,面临空间占用高和查询效率低等瓶颈。持久内存的引入虽然为 B+树提供了更高效的存储和访问方式,但在节点遍历、路径查找和结构修改等方面,仍存在性能瓶颈。与此同时,学习型索引通过结合人工智能和机器学习方法,为解决这些问题提供了新的可能。利用深度学习或强化学习模型,可对 B+树进行自适应优化,基于数据分布规律和查询负载特征调整索引结构,从而替代传统的树遍历过程,提升查询性能。未来,学习型索引可能与持久内存 B+树结合,通过机器学习模型预测高频访问的数据块,提前加载或优化数据分布,减少持久内存的访问次数,进一步提高查询效率。

2)人工智能技术的迅猛发展推动了大模型(LLM)的快速兴起,对底层计算资源提出了更高要求。尽管 AI 硬件算力持续提升,内存系统的发展却远远滞后于算力增长,导致训练与推理过程中频繁遭遇“内存墙”问题。而持久内存凭借其非易失性、高容量和字节寻址优势,结合 B+树索引为突破大语言模型中的“内存墙”问题带来了新的可能。例如,针对大模型推理效率核心组件 KV Cache[73],在未来可以采用多级分层缓存的设计,将中低频长尾信息通过 B+树索引持久化存储于 PM 中,结合 B+树的优秀范围查询特点实现历史上下文的快速回溯。

结束语　在大数据环境下,传统 B+树在查询效率、存取开销、写放大效应以及崩溃恢复等方面面临诸多挑战。相比之下,基于持久内存的 B+树利用 PM 的高速读写与非易失性特性,不仅提升了系统性能,还简化了恢复流程,增强了可靠性。本文对现有基于持久内存的 B+树索引技术进行系统综述。首先,从构建基于持久内存 B+树的挑战入手,针对持久内存的硬件特性和 B+树结构特点进行分析,总结了在 PM 上设计 B+树面临的挑战。其次,按照存储架构将现有方案划分为单一持久内存架构和混合内存架构 B+树,并且从数据一致性、并发控制优化、叶节点设计和持久化开销等方面进行进一步的分类梳理,对于不同分类下的方案,按时间顺序进行总结和对比分析。最后,总结和展望了未来持久内存 B+树的发展趋势。尽管基于持久内存的 B+树索引优化已经取得显著成效,但在性能、可靠性、可扩展性以及实际应用的

适应性方面，仍然值得后续深入研究。

参 考 文 献

[1] ZHANG H, ANDERSEN D G, PAVLO A, et al. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes[C]// Proceedings of the 2016 International Conference on Management of Data. ACM, 2016: 1567-1581.

[2] ZHANG M Z, ZHANG F, LIU Z Y. A Survey on Architecture Research of Novel Non-Volatile Memory Based on Dynamical Trade-Off[J]. Journal of Computer Research and Development, 2019, 56(4): 677-691.

[3] SHEN Z R, XUE W, SHU J W. Research on New Non-Volatile Storage[J]. Journal of Computer Research and Development, 2014, 51(2): 445-453.

[4] HELLENBRAND M, TECK I, MACMANUS-DRISCOLL J L. Progress of Emerging Non-Volatile Memory Technologies in Industry[J]. MRS Communications, 2024, 14: 1099-1112.

[5] SHU J W, LU Y Y, ZHANG J C, et al. Research progress on non-volatile memory based storage system[J]. Science & Technology Review, 2016, 34(14): 86-94.

[6] HE Y X, SHEN F F, ZHANG J, et al. Cache Optimization Approaches of Emerging Non-Volatile Memory Architecture: A Survey[J]. Journal of Computer Research and Development, 2015, 52(6): 1225-1241.

[7] QURESHI M K, SRINIVASAN V, RIVERS J A. Scalable high performance main memory system using phase-change memory technology[C]// Proceedings of the 36th Annual International Symposium on Computer Architecture. New York: ACM, 2009: 24-33.

[8] WU Z L, JIN P Q, YUE L H, et al. A Survey on PCM-Based Big Data Storage and Management[J]. Journal of Computer Research and Development, 2015, 52(2): 343-361.

[9] MAO W, LIU J N, TONG W, et al. A Review of Storage Technology Research Based on Phase Change Memory[J]. Journal of Computer Science and Technology, 2015, 38(5): 944-906.

[10] BEDESCHI F, RESTA C, KHOURI O, et al. An 8Mb demonstrator for high-density 1. 8V Phase-Change Memories[C]// 2004 Symposium on VLSI Circuits. IEEE, 2004: 442-445.

[11] 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.

[12] GAJARIA D, ADEGBIJA T. Evaluating the Performance and Energy of STT-RAM Caches for Real-World Wearable Workloads[J]. Future Generation Computer Systems, 2022, 136: 231-240.

[13] SADRI-MOSHKENANI P, KHAN M W, ISLAM M S, et al. Optoelectronic Readout of STT-RAM Based on Plasmon Drag Effect[J]. IEEE Journal of Quantum Electronics, 2021, 57(6): 1-7.

[14] HUNG J M, WEN T H, HUANG Y H, et al. 8-b Precision 8-Mb ReRAM Compute-in-Memory Macro Using Direct-Current-Free Time-Domain Readout Scheme for AI Edge Devices[J]. IEEE Journal of Solid-State Circuits, 2023, 58(1): 303-315.

[15] LI W, WANG Y, LIU C, et al. On-Line Fault Protection for ReRAM-Based Neural Networks[J]. IEEE Transactions on Computers, 2023, 72(2): 423-437.

[16] YANG J, LUO Q, XUE X, et al. A 9Mb 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]// 2023 IEEE International Solid- State Circuits Conference(ISSCC). IEEE, 2023: 1-3.

[17] LUO Y, LUC Y C, YU S. AFeRAM based Volatile/Non-Volatile Dual-Mode Buffer Memory for Deep Neural Network Training[C]// 2021 Design, Automation & Test in Europe Conference & Exhibition(DATE). IEEE, 2021: 1871-1876.

[18] IZRAELEVITZ J, YANG J, ZHANG L, et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module[J]. arXiv: 1903. 05714, 2019.

[19] YANG J, LI B, LILJA D J. Exploring Performance Characteristics of the Optane 3DXpoint Storage Technology[J]. ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 2020, 5(1): 1-28.

[20] GUGNANI S, KASHYAP A, LU X. Understanding the idiosyncrasies of real persistent memory[C]// Proceedings of the VLDB Endowment. 2020: 626-639.

[21] CHEN S, JIN Q. Persistent B+-trees in Non-Volatile Main Memory[C]// Proceedings of the VLDB Endowment. 2015: 786-797.

[22] HWANG D, KIM W H, WON Y, et al. Endurable transient inconsistency in Byte-Addressable persistent B+-Tree[C]// 16th USENIX Conference on File and Storage Technologies(FAST 18). Oakland: USENIX Association, 2018: 187-200.

[23] YOO J, CHA H, KIM W, et al. Pivotal B+tree for Byte-Addressable Persistent Memory[J]. IEEE Access, 2022, 10: 46725-46737.

[24] LI T, WANG H, SHAO A, et al. SSB-Tree: Making Persistent Memory B+-Trees Crash-Consistent and Concurrent by Lazy-Box[C]// 2022 IEEE International Parallel and Distributed Processing Symposium(IPDPS). IEEE, 2022: 70-80.

[25] ARULRAJ J, LEVANDOSKI J, MINHAS U F, et al. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory[C]// Proceedings of the VLDB Endowment. 2018: 553-565.

[26] KIM W H, KRISHNAN R M, FU X, et al. PACTree: A High Performance Persistent Range Index Using PAC Guidelines[C]// Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. ACM, 2021: 424-439.

[27] HE X, ZHANG R, TIAN P, et al. WOPE: A Write-Optimized and Parallel-Efficient B+-Tree for Persistent Memory[J]. Journal of Systems Architecture, 2024, 153: 103187.

[28] MA R X, WU F, DONG B R, et al. Write-Optimized B+ Tree Index Technology for Persistent Memory[J]. Journal of Computer Science and Technology, 2021, 36(5): 1037-1050.

[29] WANG C, CHATTOPADHYAY S. Isle-Tree: A B+-Tree with Intra-Cache Line Sorted Leaves for Non-Volatile Memory[C]// 2020 IEEE 38th International Conference on Computer Design(ICCD). IEEE, 2020: 573-580.

[30] LUO Y，JIN P，ZHANG Q，et al. TLBtree：A Read/Write-Optimized Tree Index for Non-Volatile Memory[C]// 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE，2021：1889-1894.

[31] WANG C，BRIHADISWARN G，JIANG X，et al. Circ-Tree：A B+-Tree Variantwith Circular Design for Persistent Memory [J]. IEEE Transactions on Computers，2022，71(2)：296-308.

[32] YAN W，ZHANG X J. A Concise Concurrent B+ -Tree for Persistent Memory[J]. ACM Transactions on Architecture and Code Optimization，2024，21(2)：1-25.

[33] YANG J，WEI Q，CHEN C，et al. NV-Tree：Reducing Consistency Cost for NVM-based Single Level Systems[C]// 13th USENIX Conference on File and Storage Technologies(FAST 15). USENIX Association，2015：167-181.

[34] OUKID I，LASPERAS J，NICA A，et al. FPTree：A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory[C]// Proceedings of the 2016 International Conference on Management of Data. ACM，2016：371-386.

[35] LIU J，CHEN S，WANG L. LB+Trees：Optimizing Persistent Index Performance on 3DXPoint Memory[C]// Proceedings of the VLDB Endowment. 2020：1078-1090.

[36] WANG C，CHATTOPADHYAY S，BRIHADISWARN G. Crab-tree：A Crash Recoverable B+-tree Variant for Persistent Memory with ARMv8 Architecture[J]. ACM Transactions on Embedded Computing Systems，2020，19(5)：1-26.

[37] CHEN Y，LU Y，FANG K，et al. uTree：a Persistent B+-tree with Low Tail Latency[C]// Proceedings of the VLDB Endowment. 2020：2634-2648.

[38] ZOU X，WANG F，FENG D，et al. ROWE-tree：A Read-Optimized and Write-Efficient B +-tree for Persistent Memory [C]// Proceedings of the 51st International Conference on Parallel Processing. ACM，2022：1-11.

[39] YAN W，ZHANG X J. A highly write-optimized concurrent B+-tree for persistent memory[J]. Future Generation Computer Systems，2024，155：219-230.

[40] ZHOU X，SHOU L，CHEN K，et al. DPTree：Differential Indexing for Persistent Memory[C]// Proceedings of the VLDB Endowment. 2019：421-434.

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

[42] LI Z，HE S，DANG Z，et al. CCL-BTree：A Crash-Consistent Locality-Aware B +-Tree for Reducing XPBuffer-Induced Write Amplification in Persistent Memory[C]// Proceedings of the Nineteenth European Conference on Computer Systems. ACM，2024：441-455.

[43] STONEBRAKER M. The Design of The Postgres Storage System[M]. San Francisco：Morgan Kaufmann Publishers Inc.，1987：1-19.

[44] KIESEBERG P，SCHRITTWIESER S，FRUHWIRT P，et al. Analysis of the Internals of MySQL/InnoDB B+ Tree Index Navigation from a Forensic Perspective[C]// 2019 International Conference on Software Security and Assurance (ICSSA). IEEE，2019：46-51.

[45] RODEH O，BACIK J，MASON C. BTRFS：The Linux B-Tree Filesystem[J]. ACM Transactions on Storage，2013，9(3)：1-32.

[46] SWEENEY A，DOUCETTE D，HU W，et al. Scalability in the XFS File System[C]// Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference. USENIX Association，1996：1-15.

[47] YANG C S，ZHUGE Q F，SHA X M，et al. Wear Attacks and Defense Mechanisms for Persistent In-memory File Systems[J]. Journal of Software，2020，31(6)：1909-1929.

[48] ZHANG R，LIU D，CHEN X，et al. ELOFS：An Extensible Low-Overhead Flash File System for Resource-Scarce Embedded Devices[J]. IEEE Transactions on Computers，2022，71(9)：2327-2340.

[49] YANG C，LIU D，CHEN X，et al. ReducingWrite Amplification for Inodes of Journaling File System using Persistent Memory [C]// 2019 Design，Automation & Test in Europe Conference & Exhibition(DATE). IEEE，2019：866-871.

[50] KIM D，LEE J，LIM K S，et al. An LSM Tree Augmented with B+ Tree on Nonvolatile Memory[J]. ACM Transactions on Storage，2024，20(1)：1-24.

[51] XIA F，JIANG D，XIONG J，et al. HiKV：a Hybrid Index Key-Value Store for DRAM-NVM Memory Systems[C]// Proceedings of the 2017 USENIX Annual Technical Conference. USENIX Association，2017：349-362.

[52] LEPERS B，BALMAU O，GUPTA K，et al. KVell：the Design and Implementation of a Fast Persistent Key-Value Store[C]// Proceedings of the 27th ACM Symposium on Operating Systems Principles. ACM，2019：447-461.

[53] LIUB，YE Z，HU Q，et al. HPDK：A Hybrid PM-DRAM Key-Value Store for High I/O Throughput[J]. IEEE Transactions on Computers，2024，73(6)：1575-1587.

[54] KAIYRAKHMET O，LEE S，NAM B，et al. SLM-DB：Single-Level Key-Value Store with Persistent Memory[C]// Proceedings of the 17th USENIX conference on File and Storage Technologies(FAST 19). USENIX Association，2019：191-205.

[55] CHEN Y，LU Y，YANG F，et al. FlatStore：An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory [C]// Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM，2020：1077-1091.

[56] WANG Z，SHOU L，CHEN K，et al. BushStore：Efficient B+ Tree Group Indexing for LSM-Tree in Non-Volatile Memory [C]// 2024 IEEE 40th International Conference on Data Engineering(ICDE). IEEE，2024：4127-4139.

[57] HAN X，TUCK J，AWAD A. Dolos：Improving thePerformance of Persistent Applications in ADR-Supported Secure Memory [C]// MICRO-54：54th Annual IEEE/ACM International Symposium on Microarchitecture. ACM，2021：1241-1253.

[58] ZHOU T，DU Y，YANG F，et al. EfficientAtomic Durability on eADR-enabled Persistent Memory[C]// Proceedings of the International Conference on Parallel Architectures and Compilation Techniques. ACM，2022：124-134.

[59] Intel. eADR：New Opportunities for Persistent Memory Applica-

tions [EB/OL]. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[60] MIN C, KIM K, CHO H, et al. SFS: random write considered harmful in solid state drives[C]// Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST 12). USENIX Association, 2012: 1-16.

[61] VOLOS H, TACK A J, SWIFT M M. Mnemosyne: Lightweight persistent memory[J]. ACM SIGARCH Computer Architecture News, 2011, 39(1): 91-104.

[62] CONDIT J, NIGHTINGALE E B, FROST C, et al. Better I/O Through Byte-Addressable, Persistent Memory[C]// Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles. ACM, 2009: 133-146.

[63] OWENS S, SARKAR S, SEWELL P. ABetter x86 Memory Model: x86-TSO [C]// Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. Springer, 2009: 391-407.

[64] SEWELL P, SARKAR S, OWENS S, et al. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors [J]. Communications of the ACM, 2010, 53(7): 89-97.

[65] WANG T, LEVANDOSKI J, LARSON P A. Easy Lock-Free Indexing in Non-Volatile Memory[C]// 2018 IEEE 34th International Conference on Data Engineering (ICDE). IEEE, 2018: 461-472.

[66] LEIS V, KEMPER A, NEUMANN T. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases[C]// 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 2013: 38-49.

[67] LEIS V, SCHEIBNER F, KEMPER A, et al. The ART of Practical Synchronization[C]// Proceedings of the 12th International Workshop on Data Management on New Hardware. ACM, 2016: 1-8.

[68] CHEN S, GIBBONS P B, NATH S. Rethinking Database Algorithms for Phase Change Memory[C]// Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research (CIDR). CIDR, 2011: 21-31.

[69] YANG J, KIM J, HOSEINZADEH M, et al. AnEmpirical Guide to the Behavior and Use of Scalable Persistent Memory[C]// 18th USENIX Conference on File and Storage Technologies (FAST 20). USENIX Association, 2020: 169-182.

[70] WANG Z, LIU X, YANG J, et al. Characterizing and Modeling Non-Volatile Memory Systems[C]// 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). IEEE, 2020: 496-508.

[71] LIU J, CHEN S. Initial Experience with 3DXPoint Main Memory[C]// 2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW). IEEE, 2019: 300-305.

[72] VENKATARAMAN S, TOLIA N, RANGANATHAN P, et al. Consistent andDurable Data Structures for Non-Volatile Byte-Addressable Memory[C]// 9th USENIX Conference on File and Storage Technologies (FAST 11). USENIX Association, 2011: 61-75.

[73] LIU Y, LI H, CHENG Y, et al. CacheGen: KV Cache Compression and Streaming for Fast Large Language Model Serving [C]// Proceedings of the ACM SIGCOMM 2024 Conference. ACM, 2024: 38-56.

LU Chao, born in 2001, master. His main research interest is persistent memory indexing.

ZHANG Runyu, born in 1995, Ph.D, associate professor, master supervisor, is a member of CCF (No. L9886M). His main research interests include non-volatile memories and storage systems.

(责任编辑:何杨)