计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(11): 2767−2787, 2026

分离式内存索引系统研究综述

罗旭川[1] 沈家诚[2] 周扬帆[1] 王 新[1]
[1](复旦大学计算与智能创新学院 上海 200438)
[2](昆山杜克大学 江苏昆山 215316)
(xcluo23@m.fudan.edu.cn)

Survey on Memory-Disaggregated Indexing Systems

Luo Xuchuan[1], Shen Jiacheng[2], Zhou Yangfan[1], and Wang Xin[1]
[1](*College of Computer Science and Artificial Intelligence*, *Fudan University*, *Shanghai* 200438)
[2](*Duke Kunshan University*, *Kunshan*, *Jiangsu* 215316)

Abstract Disaggregated memory (DM), as an emerging hardware architecture, has attracted extensive attention from both academia and industry in recent years. This architecture physically decouples computing and memory resources in data centers and interconnects them via high-speed networks, thereby improving resource utilization, system elasticity, and fault isolation capabilities. Distributed indexes, as the core component of memory-disaggregated storage systems, play a crucial role in efficiently organizing and maintaining data to support high-throughput, low-latency data access. However, due to the inherent resource constraints of DM, fully exploiting the benefits of resource disaggregation to maximize index performance remains challenging. A comprehensive survey of the research progress on memory-disaggregated indexing systems is presented. First, the fundamental features of DM and remote direct memory access (RDMA) are introduced, and the primary challenges in constructing high-performance indexes on DM are identified: concurrency control, read and write amplifications, and caching overhead. Subsequently, existing indexing systems are categorized into three types based on their underlying data structures: Hash indexes, tree indexes, and hybrid indexes. Representative systems in each category are analyzed to illustrate their design principles, implementation challenges, and key techniques. Finally, the main issues that require focused investigation in future research are highlighted, including cross-node cooperation, fault tolerance mechanisms, and adaptation to emerging software and hardware.

Key words disaggregated memory; distributed indexes; remote direct memory access (RDMA); resource disaggregation; data structures

摘 要 分离式内存架构作为一种新兴的硬件架构,近年来在学术界和工业界受到了广泛关注。该架构从物理上解耦数据中心的计算资源和内存资源,然后通过高速网络将分离的资源相互连接,从而提升数据中心的资源利用率、弹性和故障解耦能力。分布式索引作为分离式内存存储系统的核心组件,承担着高效组织与维护数据的关键任务,以满足用户对高吞吐、低时延数据访问的需求。然而,分离式内存架构受限于多方面的资源约束,如何充分适应资源解耦的特性并最大化发挥索引系统的性能,依然是亟待解决的重要问题。对分离式内存索引系统的研究进展进行了系统性综述。首先,介绍了分离式内存架构及远程直接内存访问(remote direct memory access, RDMA)的基本特征,并总结了构建高性能分离式内存

收稿日期: 2025-10-28; **修回日期**: 2026-07-09
基金项目: 国家自然科学基金项目(62472101)
This work was supported by the National Natural Science Foundation of China (62472101).
通信作者: 王新(xinw@fudan.edu.cn)

索引所面临的主要挑战:并发控制、读写放大与缓存开销。随后,根据数据结构类型(哈希索引、树型索引、混合索引)对现有研究进行梳理和阐述,并以多个代表性系统为实例,分析其对应的设计思路、实现难点以及关键技术。最后,指明了分离式内存索引系统进一步研究中需要着重探索的问题,包括节点协作、容错机制以及新软硬件适配。

关键词 分离式内存;分布式索引;远程直接内存访问;资源解耦;数据结构

中图法分类号 TP302

DOI: 10.7544/issn1000-1239.202550769 **CSTR:** 32373.14.issn1000-1239.202550769

数据中心作为云计算的基础设施,是数据存储与计算的核心载体,为各类云服务提供了必需的硬件资源支持。传统数据中心由单体式服务器(monolithic server)构成,即将多台单体式服务器通过数据中心网络互联,构成大规模分布式算力集群。然而,此架构存在资源利用率不足、系统弹性受限以及故障域耦合等问题。首先,单体式服务器固定的资源比例难以匹配云环境中多样化的应用需求,导致常出现资源碎片、资源搁浅(stranding)[1]等问题,整体资源利用率低。国内外许多云厂商(如阿里云[2]、谷歌[3]等)均报告了其数据中心中有超过40%的内存资源因这一原因未得到有效利用。其次,基于单体式服务器的分布式存储系统通常由每台服务器独立管理其所承载的数据,这意味着在节点拓展或缩减时需要进行数据迁移,导致系统弹性较差。最后,在单体式服务器上,一旦CPU或内存硬件出现故障,整个物理机上的所有资源都将无法使用,这增加了云服务的脆弱性。

近年来,学术界和工业界开始讨论通过使用分离式内存(disaggregated memory, DM)架构来解决传统数据中心的上述问题[4-6]。分离式内存架构在物理层面隔离了计算与内存资源,体现为将集群节点划分为计算节点和内存节点2类,并通过远程直接内存访问(remote direct memory access, RDMA)[7]等高速网络技术进行高效互联。与传统的基于单体式服务器的架构相比,分离式内存架构允许所有计算节点访问统一的共享内存池。这不仅使得内存资源能被充分利用,并且系统在弹性扩展或收缩时无需迁移原有数据,因此该架构在资源利用率和弹性方面具有显著优势。同时,由于CPU与内存解耦,计算节点和内存节点的故障能够互不影响,这提高了系统的健壮性。

分布式索引作为分离式内存架构的核心组件,承担着组织和维护分离式内存上海量数据的任务。然而,现有的分布式索引技术因依赖于远程过程调用(remote procedure call, RPC)和频繁的内存访问,在分离式内存场景下存在挑战。一方面,由于分离式内存架构中内存节点的计算能力较弱,传统基于RPC的分布式索引方案会在内存端形成计算瓶颈,难以直接适用;另一方面,基于高速网络的远端内存访问在时延和带宽上(微秒量级与Gbps量级)与内存总线(纳秒量级与GBps量级)仍存在不小的差距。因此,索引数据结构中常用的指针追逐(pointer chasing)等访存模式在分离式内存架构上会不可避免地导致高时延和低带宽利用率等问题,使得索引效率成为分离式内存系统的主要性能瓶颈。这些挑战吸引了许多研究者在近些年设计了大量分离式内存索引系统。这些系统从系统软件层面出发,通过创新的索引算法与新型的数据结构,显著优化了分离式内存的索引效率。

本文对分离式内存索引系统进行系统性综述,主要贡献有3个方面:

1)从分离式内存的架构特点和现有使用的网络互联协议出发,总结了构建高性能分离式内存索引系统面临的三大挑战,包括如何设计以计算端为中心的并发控制算法、如何解决内存端数据结构的读写放大问题以及如何优化计算端缓存空间的使用效率以避免影响索引性能。

2)根据索引所采用的数据结构类别,对现有的分离式内存索引系统进行分类介绍,包括哈希索引、树型索引以及混合索引。针对每类索引系统,以具有代表性的系统为实例,分析其对应的设计思路、实现难点与关键技术等。

3)对分离式内存索引系统的未来研究方向进行了展望,指出了研究人员未来需要着重向节点协作、容错机制和新软硬件适配这3个方向进行深入探索,以实现更高性能、更高可用的分离式内存索引系统。

1 背景介绍

1.1 分离式内存架构

分离式内存作为一种新兴的架构方案,被提出

用于从体系结构层面解决传统基于单体式服务器的数据中心的资源利用率不足、弹性差和硬件故障耦合等问题。国内外已经有部分云厂商(如阿里云[6]、微软 Azure[1] 等)开始探索和部署基于分离式内存架构的数据库和内存池以降低资源成本。

如图 1 所示，分离式内存架构在物理上将 CPU 等计算资源与 DRAM 等内存资源分离为 2 个独立的资源池：计算池和内存池。计算池中的计算节点(compute node)通常配备大量的计算资源以满足计算需求，而只拥有少量的内存作为本地缓存；与之相对，内存池中的内存节点(memory node)则配备了大量内存用于存放数据，但仅配备少量低性能计算核心用于执行一些简单任务，例如建立网络连接或分配内存空间等。这 2 类节点通过 RDMA 或计算高速链路(compute express link, CXL)[8] 等高速互联协议相互连接。由于所有计算节点都能够访问并充分利用各个内存节点上的共享内存，分离式内存架构显著提高了资源利用率。另外，分离式内存上的应用系统可以按照需求灵活地管理和分配计算池和内存池中的资源，这提高了系统弹性和硬件故障解耦能力。

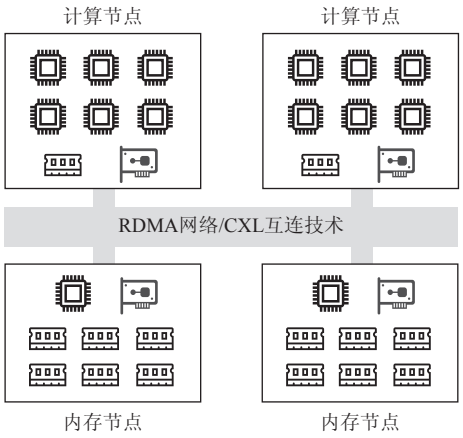


Fig. 1 DM architecture

图 1 分离式内存架构

更多关于分离式数据中心的设计背景、发展历程以及研究意义的介绍，可参考文献 [9] 中的相关综述。相比于该文献，本文主要侧重于分离式内存环境下高性能索引系统的研究进展，重点介绍相关索引结构、关键技术挑战及其优化方法。

1.2 RDMA

RDMA 网卡作为商用网卡，已经被广泛应用在了现有的数据中心中。现有数据中心的 RDMA 网络通常基于以太网部署[10]，并采用 RoCE(RDMA over converged Ethernet)协议，其端到端的时延在数微秒级别。相比于传统的内核 TCP/IP 协议，RDMA 网络的协议栈被直接卸载到了网卡硬件上，并且应用程序能够直接从用户态与 RDMA 网卡通信。这不仅避免了内核态切换的开销，还实现了零拷贝的数据传输，从而降低了时延。

RDMA 网卡为应用提供了 2 种网络原语：单边原语(one-sided verb)和双边原语(two-sided verb)。单边原语能够完全绕过接收端的 CPU 直接读写远端内存，因此十分适合用于在分离式内存架构中绕开内存端的 CPU 以直接访问数据。它包括读原语(READ)、写原语(WRITE)和一些原子原语，比如比较并交换(compare-and-swap, CAS)和获取并自增(fetch-and-add, FAA)。双边原语则需要接收端通过接收原语(RECV)预先分配数据接收缓冲池，然后发送端通过发送原语(SEND)传输数据，因此无法绕过接收端的 CPU。

RDMA 网络低时延的远程访存能力以及 CPU 旁路(CPU-bypass)的单边访存能力为分离式内存架构提供了重要机遇。然而，在高并发的数据访问负载下，基于 RDMA 的网络传输依然是分离式内存架构的主要性能瓶颈[4,11]。主要原因有 2 个：1)网卡带宽有限。近些年发布的 ConnectX-7 RDMA 网卡的带宽最高可达 400 Gbps[12]。然而，相较于本地内存总线带宽，RDMA 网卡的带宽依然存在 1 个数量级的差距。2)网卡消息处理速率有限。已经有许多研究表明[13-14]，受限于网卡硬件的消息处理能力，大量小块数据的传输会造成 RDMA 网卡的吞吐率低下。

尽管近年来已开始出现基于 CXL 互联技术的分离式内存系统及相关索引工作[15-16]，但就目前已有代表性研究而言，分离式内存索引系统仍主要围绕 RDMA 网络展开。因此，本文后续首先重点分析基于 RDMA 网络的分离式内存索引系统。基于 CXL 互联技术的代表性索引系统将在 3.4 节中进行补充介绍。

2 分离式内存索引系统的设计挑战

分离式内存架构能为数据中心带来资源利用率等各方面的好处，但受限于该架构在多方面的资源约束，在其上构建高性能的分布式索引系统面临着诸多挑战。图 2 展示了分离式内存索引系统的基本构成与相应的设计挑战：1)用户线程通过 RDMA 网络并发访问远端内存；这一部分需要解决以计算端为中心的并发控制问题。2)内存节点通过分布式索引结构维护数据地址；这一部分需要解决内存端数据结构的读写放大问题。3)计算节点通过本地缓存

加速远端索引的访问;这一部分需要解决缓存空间的使用效率问题。下面依次阐述。

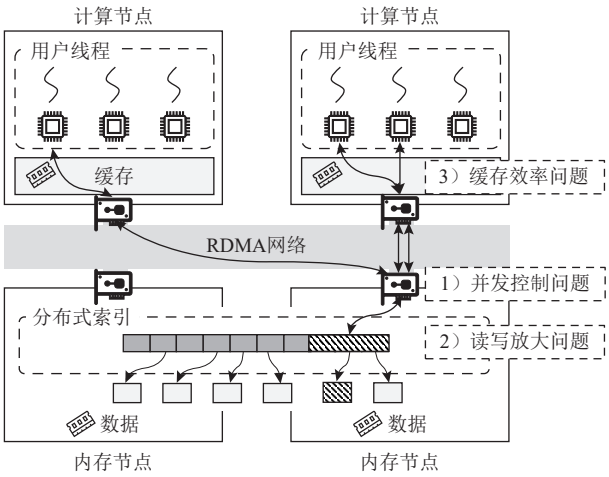


Fig. 2 DM indexing system

图 2 分离式内存索引系统

2.1 以计算端为中心的并发控制问题

在并发场景下,多个用户线程可能会同时对索引进行读写。为了避免这些并行的数据读写操作相互冲突,应用需要对索引访问进行细致的并发控制,主要包括 2 个方面:1)写写冲突(write-write conflict)。2 个写相同数据块的操作需要进行同步,以避免不同的写入操作部分完成导致最终数据处于错误状态。2)读写冲突(read-write conflict)。读操作需要感知并行的写操作,以避免读到写入数据的中间状态。

在传统的分布式索引系统中,客户端可以通过 RPC 的方式将索引的读写操作卸载到服务端的 CPU 上,从而可以在本地进行高效的并发控制。然而,在分离式内存架构中,由于内存端的计算能力受限,传统以内存端为中心的做法会形成计算瓶颈。因此,分离式内存索引系统更偏向于让计算端线程通过 RDMA 单边原语来访问内存端的索引数据结构。然而,这种以计算端为中心的并发控制算法使得应用在写写冲突与读写冲突 2 方面的处理上都存在挑战。

2.1.1 写写同步开销大

现有索引系统通常使用基于锁(lock-based)或无锁(lock-free)的方式来解决写写冲突。然而,2 种方式都需要通过远程的 CAS 原子操作实现。当 CAS 失败后,计算端线程需要重试 CAS 请求直到成功为止,这将引入大量重试 I/O,造成网卡消息处理速率瓶颈。目前的研究工作主要通过冲突缓解和冲突避免 2 个方面应对这一挑战,如表 1 所示。

在冲突缓解方面,让同一个计算节点内并发的写请求在本地预先进行同步可以有效缓解内存端的并发冲突程度。比如,Sherman[17] 提出通过锁移交(lock handover)机制来让同一个计算节点内的写线程可以共享同一次锁请求;SMART[18] 提出通过写合并(write combining)机制在计算节点内合并目标数据相同的并发写请求,从而有效减少远端写写冲突。另外,减小冲突粒度也可以有效减少不必要的写写冲突。比如,Marlin[19] 和 Deft[20] 都通过原子修改数据项的方式,使得同一个叶子节点内不同数据项之间的写操作不再冲突。然而由于原子操作存在数据长度限制,它们的方案仅针对短的数据项有效。

Table 1 Summary of Write-Write Conflict Optimization Schemes in DM Indexing Systems

表 1 分离式内存索引系统写写冲突优化方案总结

效果	技术方案	方案局限	典型系统
缓解	本地同步	/	Sherman[17], SMART[18]
缓解	减小粒度	数据长度限制	Marlin[19], Deft[20]
避免	归属划分	负载均衡问题	DEX[21], DINOMO[22]

注:“/”表示无明显局限性。

在冲突避免方面,DEX[21] 和 DINOMO[22] 采用了归属划分(ownership partitioning)方案,让每个计算节点负责特定的数据范围,这样计算节点对内存端的数据访问就不会存在冲突。然而这一方案会导致计算节点总是与请求的数据键绑定,而无法灵活地进行请求调度以实现负载均衡。

2.1.2 读写校验设计复杂

在真实的云存储系统中,大多数的应用负载都是读密集的[23-24],因此现有分离式内存系统通常设计无锁读机制以提升读操作性能。实现无锁读主要有 2 种做法:1)将所有写操作实现为无锁的。无锁的写操作保证读写冲突中的数据都是由原子操作进行修改,因此读操作总是能读到写操作前后正确的值。这类做法对索引设计要求较高,不常被使用。2)采用乐观并发控制。写操作通过锁同步,读操作在读到数据之后校验数据的一致性。这类方法在分离式内存上被广泛应用。

如表 2 所示,对于一块远端数据块,现有正确的乐观读写校验方案主要有 3 种。1)在数据块中嵌入 1 个校验码(checksum),如循环冗余校验(cyclic redundancy check, CRC)。写线程在写入新数据时重计算和更新校验码;读线程在读到数据后需要判断读到的数据与校验码二者是否匹配,若不匹配则进行重读。2)为数据块额外维护 1 个普通版本号(version)。写线程在写入新数据的同时更新版本号;读线程先

读版本号再读数据，最后再读1次版本号以判断版本号是否改变，若改变则进行重读。该方案在分离式内存架构上会引入多次网络往返时延(round-trip time, RTT)，因此通常不被采用。3)缓存行版本号(cache line version)。该方案由微软研究院提出[28]，它在数据块的头部和其所覆盖的每个缓存行的头部都插入1个版本号。基于x86架构下直接内存访问(direct memory access, DMA)的缓存一致性，所有缓存行版本号的一致即能保证这些缓存行拼接出的内容是整体一致的。因此，写线程在写入新数据时需要更新所有版本号；读线程读到数据后需要判断所有缓存行版本号是否一致，若不一致则进行重读。

Table 2 Summary of Read-Write Check Schemes in DM Indexing Systems

表2 分离式内存索引系统读写校验方案总结

技术方案	方案局限	典型系统
无锁	设计困难	RACE[25]，SMART[18]
校验码	计算开销	RCuckoo[26]，SMART[18]等
普通版本号	网络开销	/
缓存行版本号	内存开销	Deft[20]，CHIME[27]等
首尾版本号	不正确	Sherman[17]，Marlin[19]

注："/"表示未被典型系统采用。

然而，在索引结构中，上述乐观并发控制的设计是复杂的。一方面，除了简单的数据项写操作，索引中存在如节点分裂、哈希扩容等复杂的结构修改操作(structural modification operation, SMO)，使得读写校验需要应对多种复杂情形。许多索引系统甚至需要同时采用多种校验方案来兼顾各种并发情形。比如，SMART[18]对内部节点采用无锁方案，而对叶子节点采用校验码方案；Deft[20]不仅采用缓存行版本号保证叶子节点上数据的一致性，还通过比对父子节点上的版本号来让读操作感知B+树的分裂操作。另一方面，读写校验方案通常会考虑利用RDMA网卡硬件特性(比如顺序写[29])来避免额外的RTT。然而，对硬件特性不充分的理解会导致一些隐蔽的错误。例如，Sherman[17]提出的首尾版本号技术被证明了是不正确的[30]，并影响了采用该技术的其他索引系统。

2.2 内存端数据结构的读写放大问题

在分离式内存架构中，由于内存端CPU的计算能力有限，索引操作一般通过RDMA单边原语实现。由于RDMA网络相比本地内存访问的时延更高，计算端线程通常采用大块读的策略以避免频繁的网络往返。这种访存模式导致了读写放大的问题。由于RDMA网络带宽有限，这个读写放大会导致低带宽利用率，降低系统吞吐。

以B+树的读放大为例，当计算端的用户线程需要访问数据项时，用户线程需要逐层、完整地读取从根节点到目标叶子节点路径上的所有的内部节点，以及最后包含了数据项的完整的叶子节点。考虑将内部节点全部进行缓存，那么在1 KB的叶子节点中搜索1个16 B数据项将会造成64倍的读放大。这意味着系统的带宽利用率只有1/64。目前的研究工作主要通过内存端数据结构改造和计算端元数据缓存2种方式来应对这一挑战，如表3所示。

Table 3 Summary of Read/Write Amplification Optimization Schemes in DM Indexing Systems

表3 分离式内存索引系统读写放大优化方案总结

方式	技术方案	方案局限	典型系统
改造	基数树	缓存开销	SMART[18]
改造	混合结构	未彻底优化	Deft[20]，CHIME[27]
缓存	元数据表	缓存开销	SepHash[31]，CHIME[27]

在数据结构改造方面，通过选择读写放大更小的数据结构作为索引结构，或者设计新型的混合索引结构可以有效减小读写放大。比如，SMART[18]由于基数树的读写放大更小，专门设计了针对分离式内存优化的基数树索引。然而基数树引入了较大的缓存开销，其缓存开销为B+树类索引的16倍以上[27]。Deft[20]和CHIME[27]用哈希表替换B+树的叶子节点，从而将读写放大范围从叶子节点大小缩小到数个哈希桶的大小或1个跳房子哈希邻域(neighborhood)的大小。

在元数据缓存方面，通过在计算端缓存远端数据结构的空位信息或部分数据项的地址信息，可以让计算端线程得以更精细地读取数据。比如，SepHash[31]通过在计算端缓存1个指纹表来减少其大数据段的读放大问题；CHIME[27]通过缓存热数据项的精细地址，来进一步减小跳房子哈希邻域的读写放大。然而，这类缓存的方案需要利用有限的计算端内存，因此需要对缓存大小进行额外控制。

2.3 计算端缓存空间的使用效率问题

分离式内存索引系统通常会通过计算端缓存来提升索引效率。特别是对于树型索引，计算端通常会将部分内部节点缓存在本地。用户线程可以先搜索本地缓存来节省一部分远程索引遍历的开销。这可以有效减少基于RDMA单边原语的索引遍历的RTT次数，减少数据访问时延。然而，由于计算端的

内存资源有限,索引系统需要尽可能提高索引的缓存效率以最大化缓存带来的收益。目前的研究工作主要通过数据结构调整和替换算法改进2个方面来应对这一挑战,如表4所示。

Table 4 Summary of Cache Efficiency Optimization Schemes in DM Indexing Systems

表4 分离式内存索引系统缓存效率优化方案总结

维度	技术方案	方案局限	典型系统
结构	大叶子节点	读放大	Dolphin[32] 等
结构	机器学习	精度敏感	ROLEX[33], XStore[34]
结构	哈希表辅助	维护开销	Sphinx[35]
算法	路径缓存	效果局限	DEX[21]

在数据结构调整方面,调整节点大小可以有效降低树型索引的缓存开销。比如,Dolphin[32]指出可以通过使用大的B+树叶子节点来减少内部节点数量,从而减少缓存开销。然而,增大叶子节点大小的同时也加重了B+树的读放大问题。另外,用缓存友好的数据结构替换或辅助内部节点也可以减少索引遍历开销。比如,ROLEX[33]提出用机器学习模型替换分离式内存B+树的内部节点,这样一来计算端只需要缓存小模型;Sphinx[35]通过前缀-地址映射的哈希表来加速分离式内存基数树的内部节点遍历过程,计算端只需要缓存哈希表信息。然而,这些方法或多或少会引入额外的网络开销:ROLEX模型的精度不足会导致目标数据项的可能位置落在多个叶子节点上;Sphinx对于写请求需要额外的网络I/O用于更新辅助的哈希表。

在替换算法方面,通过选择更加合适的内部节点来缓存也能帮助提升缓存效率。DEX[21]使用了路径感知的缓存策略,保证对于每个缓存的节点,其父节点也被缓存。这类方法相比结构调整的方法效果更加局限,因为其无法帮助索引结构降低缓存开销。

3 分离式内存索引系统的研究进展

分离式内存索引系统在享受分离式内存架构带来的高资源利用率等好处的同时,通过精细的索引系统软件技术,实现高效的并发控制、读写操作和缓存管理。我们将现有的分离式内存索引系统分为3类,包括哈希索引、树型索引以及混合索引。本节将依次介绍这3类分离式内存索引系统,并详细分析典型的系统实例。表5对这3类索引在多方面进行了对比。

Table 5 Comparison of Three Types of Indexing Systems on DM

表5 分离式内存上3种索引系统的对比

维度	哈希索引系统	树型索引系统	混合索引系统
数据结构	多路关联哈希表、布谷哈希表、分层哈希表等	B+树、基数树、日志结构合并树、跳表等	基数树与哈希表、B+树与小模型、基数树与B+树、B+树与哈希表等
范围查询	不支持	支持	支持
并发复杂程度	低	中	高
读写放大程度	中	低至高(取决于结构选择)	低至高(取决于结构组合)
缓存空间开销	低	中至高(取决于结构选择)	低至中(取决于结构组合)
典型系统	RACE[25], RCuckoo[26], SepHash[31], Outback[36]	FG[37], Sherman[17], Marlin[19], DEX[21], SMART[18], GECKO[38], dLSM[39], Mariana[40]	Sphinx[35], ROLEX[33], Dolphin[32], HT-tree[41], Deft[20], CHIME[27]

3.1 分离式内存哈希索引

分离式内存哈希索引通过哈希表来维护内存端的数据,为计算端的用户线程提供高吞吐、低时延的点查询。RACE和RCuckoo是2个具有代表性的分离式内存哈希索引系统,它们分别展现了封闭寻址(closed addressing)和开放寻址(open addressing)两类哈希冲突解决算法在分离式内存上的实现方案。本节首先着重介绍这2个代表性系统,然后对剩余的哈希索引系统进行简要归纳。

3.1.1 分离式内存上的多路关联哈希索引

RACE[25]由华为云提出,是首个针对分离式内存优化的哈希索引系统,被广泛应用在多个分离式内存存储系统中[42-43]。在分离式内存架构中,为了避免陷入内存端的计算瓶颈,索引系统需要通过RDMA单边原语来完成查询和修改操作。这带来了3个方面的问题:1)内存利用率高的哈希冲突算法通常通过数据移动来为新插入的数据腾出空间(比如布谷哈希[44]、跳房子哈希[45]等)。它们在分离式内存上会引入多次远端内存读写,造成高时延。2)传统的哈希表普遍采用基于锁的并发控制。然而基于RDMA单边原语的远端锁的网络开销巨大。3)当哈希表装满时,需要对其进行扩容。然而哈希表扩容会使得

其他用户线程难以并行访问。针对上述挑战，RACE设计了对RDMA单边原语友好的哈希表结构，实现了无锁的并发控制并设计了高效的扩容方案。

图3展示了RACE的基本架构。RACE哈希表由1个目录(directory)和多个子表(subtable)组成。为了减少额外的单边READ操作，RACE将目录完全缓存在计算端。用户线程通过本地内存访问目录以找到目标子表，再通过RDMA单边原语访问远端子表。

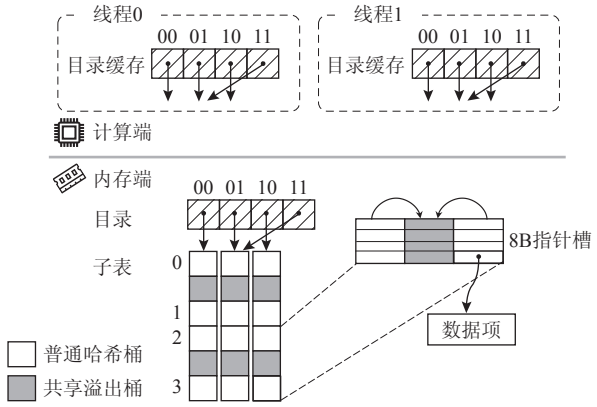


Fig. 3 Basic architecture of RACE

图3 RACE的基本架构

RACE的每个子表是个包含了多个哈希桶的数组。为尽可能提高哈希表的最大负载因子(load factor)，RACE在子表结构上应用了3项技术：1)多路关联。具体而言，每个哈希桶里包含多个指针槽(slot)，K路关联表示每个哈希桶里有K个槽。哈希值相同的数据可以存放在同一个哈希桶内的不同槽中，而不需要进行移动。每个哈希桶里的指针可以被1次单边READ操作一起读取。2)2次选择[46]。RACE使用2个无关的哈希函数，让每个数据项可以选择2个哈希桶中的任意1个进行存放。在查询数据时，RACE通过门铃合并(doorbell batching[11])发出2个单边READ操作并行地读取2个哈希桶。3)溢出存放[47]。对于每对相邻哈希桶，RACE在其间插入1个溢出桶以存放溢出的项。溢出桶可以和与其相邻的主哈希桶被1次单边READ操作一并读取。

RACE哈希表在哈希桶内的每个槽中存放数据的指纹(fingerprint)、长度和地址。指纹是个8 b哈希值，用于快速判断目标数据是否位于该槽指向的内存地址上；长度字段同样为8 b，长度单位为64 B，因此可覆盖的最大数据块长度为16 KB。当键值项超过16 KB时，RACE将超出部分写入后续数据块，并以链式指针连接到首块；数据块头部还记录键长和值长，用于在读写路径中确定数据边界。由于x86_64系统上虚拟地址通常仅低48 b有效，RACE可将指纹、长度和地址压缩到单个8 B指针槽中，使其能够被RDMA网卡原子修改，从而为无锁操作提供数据结构基础。

RACE的查询、插入、更新和删除操作都是无锁的。用户线程首先会通过单边READ操作读取对应的哈希桶；对于插入和更新操作，用户线程会在读取桶的同时将新数据写到远端内存。在这之后：1)对于查询操作，若存在指纹匹配的项，则用户线程会进一步读取实际存放数据的内存块来判断其是否包含目标数据；若没有匹配的项，说明目标数据不存在。2)对于更新和删除操作，用户线程通过单边CAS操作将远端哈希桶内旧数据的指针修改为新数据的地址或者空指针。旧数据项的内存将在后台清空。3)对于插入操作，用户线程会在读取的桶中寻找是否有空槽，有则使用原子的CAS操作将新数据的地址插入远端哈希桶的对应位置，否则触发远程扩容操作。为了避免相同的数据被并行用户线程重复地插入哈希桶，RACE在CAS操作后会重读1次哈希桶进行检查。4)对于远程扩容操作，RACE先对位于内存端的目录加锁，以防止其他用户线程并发扩容，但不会阻塞其他并行的数据读写操作。用户线程创建新的子表后更新目录中的指针，然后基于多个原子CAS操作将重新哈希后的数据项逐个移动到新的子表中，这一过程不会影响其他用户线程对旧的子表内的数据进行并行访问。由于RACE将目录完全缓存在计算端，当其他计算节点触发远端哈希表扩容并更新目录后，本地缓存的旧目录可能失效，导致线程依据旧目录访问到错误的子表。为此，RACE在每个哈希桶中存放用于标识其目录归属的元数据。用户线程读取哈希桶后，会校验该桶是否与本地目录匹配；若不匹配，则从内存端重新读取最新目录并重试查询操作。

表6总结了RACE对第2节所述的3点挑战的处理情况。在并发控制上，RACE设计了无锁的数据结构，避免了远端锁的开销；无锁结构也使得读操作可以无锁进行而不需要额外校验。在读写放大上，RACE为了提升最大负载因子采用了多路关联、2次选择和溢出存放等技巧。这些技巧在一定程度上引入了读放大问题。当使用4路关联时，RACE的读放大倍数为16；并且，多路关联使得RACE的插入操作需要进行重读来检查是否有重复数据被插入。在缓存开销上，RACE只需要缓存目录结构，由于哈希表目录通常仅包含数百条目录项，而每条目录项仅占

Table 6 RACE's Solutions to the Challenges of DM Indexes

表 6 RACE 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	无锁结构	避免锁开销与读写校验
读写放大	多路关联、2 次选择	存在读放大
缓存开销	缓存目录	缓存开销小

用数字节, 因此缓存开销仅为 KB 量级[25]。

3.1.2 分离式内存上的布谷哈希索引

RCuckoo[26] 由加州大学圣迭戈分校提出, 是一个基于布谷哈希的分离式内存索引系统。RCuckoo 指出, 在高并发场景, 无锁的乐观方案会相比基于锁的悲观方案表现更差; 另外, 无锁的哈希表设计使得长度超过原子修改限制(如 8 B)的数据项无法内联(in-line)在哈希表中, 这会导致读操作产生额外的 RTT 开销。因此, RCuckoo 设计了一种基于锁的分离式内存哈希表, 利用布谷哈希算法提升最大负载因子, 同时通过约束 2 次哈希的位置局部性来减少 I/O 数量。

图 4 展示了 RCuckoo 的基本架构。RCuckoo 由 1 个锁表(lock table)和 1 个布谷哈希表组成, 图中哈希表的每行是 1 个哈希桶。用户线程首先计算目标数据的 2 个哈希值, 然后访问 2 个值对应的 2 个哈希桶, 或者获取相应的锁。

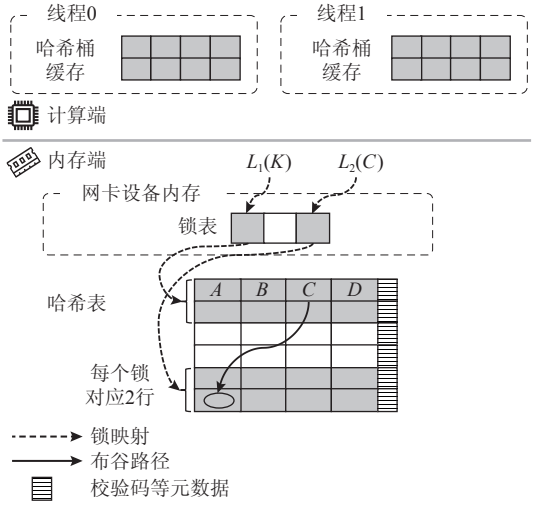


Fig. 4 Basic architecture of RCuckoo

图 4 RCuckoo 的基本架构

为了优化 RTT 次数, RCuckoo 将短的数据可选地内联在哈希表中。具体而言, 每个哈希桶内所有条目的格式在建表时固定, 单个条目可表示为 2 类: 一类为内联条目(inline entry), 直接在槽位中存放短键值数据; 另一类为扩展条目(extent entry), 在槽位中存放键、48 b 指针与扩展长度, 数据的值则放在指针所指向的额外内存区域。每个条目中的最低数据位用于标识条目类型, 默认采用内联模式。为减少插入时的分配争用, 扩展条目所指向的内存区域并非属于共享内存池, 而是放在预分配的、按客户端划分的额外 RDMA 注册内存区域中。

为了减小网卡和内存间的 PCIe 开销, RCuckoo 将锁表结构存放在了网卡的设备内存(device memory)上。在高并发情形, 设备内存吞吐约为主存的 3 倍, 且由于不经过主存 PCIe 往返, 访问时延也更低。然而在 ConnectX-5 网卡中, 设备内存容量仅有约 256 KB, 这直接限制了可容纳的物理锁表的规模。因此, RCuckoo 采用单比特锁设计, 并引入了虚拟锁表机制: 将多个逻辑锁映射到同 1 个物理锁。这种多对 1 映射会额外引入伪冲突, 但换来了对任意锁表规模的支持。启用虚拟锁时, 客户端在插入路径上会先把目标数据行映射到虚拟锁, 再映射到物理锁。

RCuckoo 的另一个技术核心在于通过使用 2 个存在依赖关系的函数, 来控制 2 个哈希桶的位置尽可能接近。具体而言, RCuckoo 使用了 3 个哈希函数 h_1, h_2, h_3。数据 K 的 2 个布谷哈希桶的位置将被计算为

$$\begin{aligned} L_1(K) &= h_1(K) \bmod T, \\ L_2(K) &= (L_1 + h_2(K) \bmod \lfloor f^{f+Z(h_3(K))} \rfloor) \bmod T, \end{aligned} \tag{1}$$

其中 T 是哈希表的行数, $Z(x)$ 是 x 的二进制表示中尾随 0 的个数, f 是控制局部性程度的超参数。由于尾随 0 的个数呈几何分布, 因此式(1)既使得位置 L_2 与 L_1 的距离上限(即 $\lfloor f^{f+Z(h_3(K))} \rfloor$)有一半概率能被限制在 f^f, 同时又保留了一定的突破限制的机会以维持布谷哈希解决冲突的能力。这样, 布谷哈希的 2 次 RDMA 单边请求就可以有概率被合并为 1 次。这不仅可以减少读写开销, 还能优化上锁次数。

RCuckoo 的查询操作是无锁的, 插入、更新和删除操作是基于锁的。具体而言: 1)对于查询操作, 用户线程首先计算 2 个哈希桶的位置。如果 2 个位置比较接近, 则直接通过 1 次单边 READ 操作读取数据; 否则回退到 2 次 READ 操作。用户线程通过校验码验证哈希桶是否正在被写, 若是则重读。2)对于更新和删除操作, RCuckoo 在确定 2 个哈希桶的位置后, 通过并行的单边 CAS 和 READ 操作同时对 2 个哈希桶进行上锁和读取。若目标数据存在, 则再通过并行的单边 WRITE 和 CAS 操作对 2 个哈希桶进行修改和解锁。和查询操作一样, 如果 2 个哈希桶位置接近, 则网络 I/O 次数可以减半。3)对于插入操作, RCuckoo 在计算端维护了 1 个专门用于辅助插入操作的哈希表缓存。这个缓存大小被限制为 64 KB, 只

会缓存部分的哈希桶，并会随着各种操作的结果而更新。用户线程首先利用本地哈希表缓存确定可能的布谷路径(cuckoo path)，然后并行地对路径所包含的各个哈希桶进行上锁和读取，并更新缓存。接着，用户线程在更新后的缓存中再次搜索，确认是否存在一个候选路径(可以是初始的路径或只包含当前已上锁的哈希桶的新路径)。若存在，则通过单边WRITE原语执行数据插入；否则释放所有锁，并重试插入操作。

表7总结了RCuckoo在第2节所述的3点挑战上的表现。在并发控制方面，RCuckoo选择使用基于片上锁的并发控制来优化更高并发场景的冲突程度，并通过约束2个哈希桶的位置局部性来减少上锁次数。在读写放大方面，布谷哈希需要用户线程读取2个哈希桶，而每个哈希桶在实现上存放8个槽，因此其存在至少16倍的读放大。并且，在哈希表接近饱和时，较长的布谷路径会导致较大的写放大。当负载因子为90%时，RCuckoo的插入时延约为插入空表时的2倍[26]。在缓存开销方面，RCuckoo只需要64 KB的哈希表缓存，因此开销较小。

Table 7 RCuckoo's Solutions to the Challenges of DM Indexes

表7 RCuckoo对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	片上锁、局部性约束	优化冲突、降低锁开销
读写放大	布谷哈希	存在读写放大
缓存开销	固定大小的缓存	缓存开销小

3.1.3 分离式内存上的其他哈希索引

除了RACE和RCuckoo，还有一些研究工作提出了优化的分离式内存哈希索引。SepHash[31]索引设计了分层的哈希表结构，将每个子表替换为了2层的解耦数据段(当前数据段和主数据段)。用户线程的并发写操作可采用仅追加(append-only)的方式写入当前数据段；当前数据段写满后再合并到主数据段中；主数据段写满后则进行分裂。2层结构使得哈希扩容造成的数据移动可以批量进行，优化了扩容效率，但同时使得读操作需要额外的RTT来确认数据项所在的层。Outback[36]索引探索了基于完美哈希(perfect hashing)的方案。它使用了一种动态完美哈希算法Ludo[48]，将其中计算密集的模块放在计算端进行，而访存密集的模块则放在内存端并通过RPC的方式进行。内存端维护了溢出缓存用于暂时存放哈希冲突的数据，当溢出数量超过限制时则进行哈希扩容(即重计算完美哈希函数)。然而在高并发场景，RPC操作可能会造成内存端的计算瓶颈。分离式内存混合索引CHIME[27]中包含了完全基于RDMA单边原语实现的跳房子哈希表设计。它通过嵌入在锁中的位图来维护哈希表中的空闲槽位置。这使得用户线程可以在上锁的同时提前确认插入操作将会影响的内存范围，并通过1次单边READ和1次单边WRITE操作实现数据插入时的数据移动效果。然而和布谷哈希表类似，跳房子哈希表在接近饱和时，数据移动范围会变大，导致插入效率降低。表8对上述分离式内存哈希索引进行了总结对比。

Table 8 Comparison of DM Hash Indexing Systems

表8 分离式内存哈希索引系统对比

系统	哈希算法	方案局限
RACE[25]	多路关联哈希	插入操作需要重读检查
RCuckoo[26]	布谷哈希	接近饱和时插入开销大
SepHash[31]	分层哈希	查询需要额外RTT
Outback[36]	完美哈希	需要利用内存端CPU
CHIME[27](叶子节点)	跳房子哈希	接近饱和时插入开销大

3.2 分离式内存树型索引

分离式内存树型索引不仅能为用户提供高效点查询，还能支持范围查询操作。在各类树型数据结构中，B+树作为最典型的一种结构，被广泛应用于各类数据库系统中[6]，也因此被首先考虑用于分离式内存系统；在这之后，各种其他树结构也被逐步探索用于分离式内存系统。本节首先以Sherman和SMART为例，分别着重介绍基于B+树和基数树这2类代表性结构的分离式内存索引系统，然后再对其他的树型结构方案进行简要介绍。

3.2.1 分离式内存上的B+树索引

Sherman[17]由清华大学提出，是首个针对分离式内存进行优化的B+树索引系统，吸引了后续众多针对分离式内存的树型索引设计。分离式内存要求用户线程完全通过RDMA单边原语来对内存端的分布式B+树索引进行读写操作。尽管在Sherman之前已经有基于RDMA单边实现的B+树索引FG[37]，但其未解决以下由分离式内存带来的性能挑战：1)B+树需要通过基于单边CAS原语的远端锁来进行并发控制。然而，在高并发、高倾斜负载下，大量用户线程的锁请求冲突会让性能急剧下降。2)传统B+树的数据有序的叶子节点结构会导致写操作通常要求将叶子节点重新排序再全部写回，这会导致严重的写放大问题。3)基于RDMA单边原语的树型索引的写操作需

要多次 RTT。例如,对数据项的更新操作包括加锁、读树节点、写回修改后的树节点、释放锁等。这些操作分别需要 1 次 RTT,导致较高的时延。为此,Sherman 设计了分层片上锁(hierarchical on-chip lock, HOCL)优化并发冲突,设计了数据无序的叶子节点和 2 级版本号(two-level versioning)消除写放大,提出了精细的 RDMA 请求合并设计来减少 RTT 次数。

图 5 展示了 Sherman 的基本架构。Sherman 将 B+树的节点均匀分布在各个内存节点中,每个内存节点上的所有锁被解耦成 1 个全局锁表(global lock table),放在了网卡设备内存中以减小 PCIe 开销。每个计算节点缓存了部分的内部节点以减小单边的树遍历开销。用户线程首先查询本地缓存再通过 RDMA 单边原语访问远端 B+树。

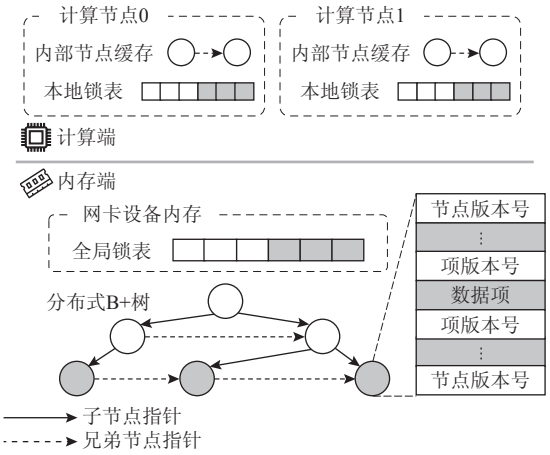


Fig. 5 Basic architecture of Sherman

图 5 Sherman 的基本架构

Sherman 在每个计算节点上维护了本地锁表用于实现 HOCL 机制:1)当用户线程需要对某个远端树节点加锁时,首先尝试获取对应的本地锁。若本地加锁失败,则没有必要进一步尝试请求远端锁,这缓解了远端锁的 CAS 并发冲突。2)当用户线程需要释放远端锁时,首先会检查本地是否有其他线程请求相同的锁。若有,则可以将这个远端锁的持有状态直接移交给其他线程,这避免了当前用户线程基于 WRITE 原语的锁释放开销和其他用户线程基于 CAS 原语的锁请求开销。

Sherman 采用了无序的叶子节点结构,并通过 2 级版本号对树节点和每个数据项单独维护了读写一致性。具体而言,Sherman 在每个叶子节点的头部和尾部维护了 1 组版本号,在叶子节点内部每个数据项的头部和尾部也分别维护了 1 组版本号。当叶子节点发生分裂或合并时,仅更新节点首尾的版本号;当用户线程更新数据项时,仅更新数据项首尾的版本号。读线程通过检查节点首尾的版本号感知是否有并行的节点分裂或合并操作,通过检查目标数据项首尾的版本号来感知该数据是否正在被修改。这样一来,Sherman 在修改数据项时,只需要写回被修改的部分,而不需要写回整个叶子节点,从而降低了写放大。需要注意的是,由于 RDMA 网卡硬件并不具备顺序读的特性,首尾版本号的设计是不正确的[30];但 2 级版本号的设计思路是正确的。这启发了后续工作进一步改进 B+树并发控制算法的正确性,例如 CHIME[27] 和 Deft[20]。

基于上述核心技术,Sherman 设计了精细的 B+树读写操作。具体而言,用户线程无锁地遍历内部节点,在定位到叶子节点后:1)对于插入、更新或删除操作,用户线程首先基于 HOCL 机制通过单边 CAS 原语对目标叶子节点上锁;通过单边 READ 原语读取整个节点;用单边 WRITE 原语写新数据到空位、更新数据项或清空数据项;最后通过 1 次单边 WRITE 原语解锁。这里 Sherman 利用 RDMA 协议中可靠连接(reliable connection)队列对(queue pair)的顺序写特性[29],将后 2 个单边 WRITE 操作合并发出,以此减少了 1 次 RTT 开销。若触发了 B+树的分裂或合并操作,则通过多次单边操作分别修改整个叶子节点、递归地上锁和更新父节点。同样,每个节点的修改和解锁操作也可以合并以减少 1 次 RTT 开销。2)查询操作是无锁的,用户线程通过单边 READ 原语读取整个叶子节点,并通过 2 级版本号检查读写一致性。若版本号不一致则重读。对于范围查询,用户线程则通过并行的单边 READ 原语同时读取多个叶子节点。

表 9 总结了 Sherman 在第 2 节所述的 3 点挑战上的表现。在并发控制方面,Sherman 提出的 HOCL 机制通过在计算节点内进行锁移交,有效减少了远端锁的并发冲突;并且远端锁放置在网卡设备内存上能降低锁开销。在读写放大方面,Sherman 通过 2 级版本号让用户线程可以只写回修改的数据项,因此在更新数据项时,其写放大倍数约为 1;但其在首尾版本号的设计上存在正确性问题[30]。在缓存开销上,增大 B+树的叶子节点可以减少内部节点数量;当叶子节点扇出大小为 64 时,Sherman 装载 6 000 万个数据项仅需 24 MB 的计算端缓存[27]。

3.2.2 分离式内存上的基数树索引

SMART[18] 由复旦大学提出,是分离式内存上的首个基数树索引系统。SMART 指出,B+树固有的读

Table 9 Sherman's Solutions to the Challenges of DM Indexes

表 9 Sherman 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	分层片上锁	减少并发冲突、降低锁开销
读写放大	2 级版本号	降低写放大，但正确性不足
缓存开销	缓存内部节点	缓存开销较小

放大问题会在分离式内存架构上导致性能低下。当需要查询数据项时，用户线程需要读取 B+树搜索路径上所有完整的内部节点和叶子节点。这种读放大会浪费额外的 RDMA 带宽，降低 I/O 效率。基数树，特别是动态基数树[49]，由于其内部节点仅存放部分键(partial key)，并且每个叶子节点仅包含 1 个数据项，因此相比 B+树，它的读写放大更小，更适合作为分离式内存架构上的树型索引。

在分离式内存上构建基数树有 3 个方面的挑战：1)并发控制开销。传统基数树的并发控制是基于锁的[50]，而基于 RDMA 单边原语的远端锁开销巨大。2)网卡消息处理速率瓶颈。基数树读写放大很小，因此不会受限于网络带宽。然而，RDMA 网卡的消息处理速率会限制基数树达到更高的吞吐率。3)缓存校验问题。基数树有许多复杂的优化技术，比如路径压缩、节点大小动态调整等[49]。这些技术会使得内存端树结构发生改变，导致计算端的内部节点缓存失效。针对上述问题，SMART 设计了混合的基数树并发控制方案，提出了读代理和写合并技术以优化吞吐率，并设计了专门的缓存校验机制以更新失效的缓存项。

图 6 展示了 SMART 的基本架构。和 B+树一样，SMART 将基数树节点均匀地分布在各个内存节点上。SMART 的内部节点采用无锁的设计；叶子节点采用的则是基于锁的设计。为了实现缓存校验，每个树节点内都维护了 1 个指向父节点的反向指针；如果读到的反向指针与父节点地址不匹配，说明远端树结构发生了改变。每个计算节点缓存了内部节点以减少树遍历的 RTT。用户线程首先查询本地缓存再通过 RDMA 单边原语访问远端基数树。

传统基数树的内部节点包含 1 个部分键数组和 1 个指针数组；部分键与指针一一对应。用户线程首先找到与目标数据相匹配的部分键，然后沿着其对应的指针继续读取下个树节点。SMART 利用 x86 架构下指针前 16 b 的空闲空间，将部分键直接嵌入到对应的指针中。因此，部分键和对应的指针就可以通过 1 次单边 CAS 原语原子地修改。这种内部节点结构可以被无锁地读写，因此避免了粗粒度的内部节点锁。对于叶子节点，SMART 指出基于异地更新的叶子节点无锁修改会造成叶子节点地址频繁变动，导致计算端产生缓存抖动(cache thrashing)的问题。因此，SMART 对叶子节点采用基于锁的并发控制，并通过校验码进行读写校验。不同于 B+树的带宽瓶颈，基数树的性能瓶颈为网卡的消息处理速率。因此 SMART 选择将锁放置在叶子节点尾部而非网卡设备内存中；这样就可以利用 RDMA 网卡的顺序写特性，通过 1 次 WRITE 操作在写回数据的同时将锁释放。

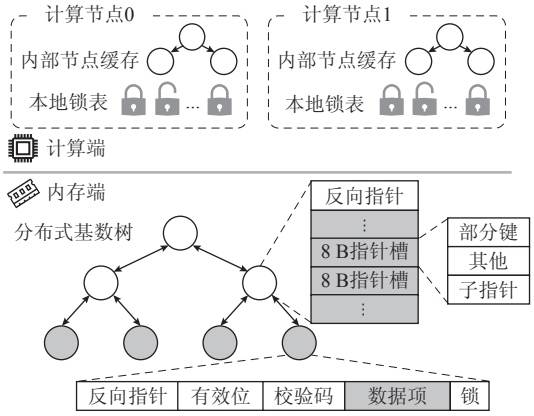


Fig. 6 Basic architecture of SMART

图 6 SMART 的基本架构

为了进一步优化吞吐率，SMART 在每个计算节点上维护了 1 个本地锁表结构以实现读代理和写合并技术。每个用户线程会尝试获取与目标数据项对应的本地锁，取锁成功的线程会被指派为代理线程；取锁失败的线程则在本地等待。对于并发的读请求，由代理线程执行远端操作，并将结果共享给其他等待的本地线程，从而减少了冗余的读 I/O。对于并发的写请求，SMART 则将现代处理器中的写合并技术[51]迁移到了分离式内存架构上：通过本地的写合并缓冲池维护计算节点内用户线程的共识结果，由代理线程来获取远端锁，并把共识的结果写回到远端内存。这不仅减少了冗余的写 I/O，还减少了远端锁的并发冲突。

基于上述核心技术，SMART 设计了精细的基数树读写操作。具体而言，被选举出的代理用户线程无锁地遍历内部节点，在定位叶子节点后：1)对于查询操作，该用户线程通过 1 次单边 READ 直接读取叶子节点内的数据项并通过校验码来检查读写一致性；检查读取的反向指针是否指向缓存内的父节点，若

不是则说明缓存失效,该缓存项将被剔除,查询操作将重试。对于范围查询操作,SMART 和 Sherman 类似,通过并行的单边 READ 原语同时读取和校验多个叶子节点。2)对于插入或更新操作,用户线程首先和查询操作一样读取和校验叶子节点。如果叶子节点内的数据与目标数据匹配,则用户线程通过单边 CAS 原语对叶子节点上锁然后通过单边 WRITE 原语就地更新数据项并解锁。如果数据不匹配,则用户线程将新数据通过单边 WRITE 原语写到新的位置,然后计算 2 个数据的最长匹配前缀,将新数据的地址通过单边 CAS 原语无锁地插入对应的内部节点,并在必要时进行相应的节点分裂。3)对于删除操作,用户线程直接通过单边 CAS 原语清空内部节点中指向目标叶子节点的指针,并在叶子节点上标记删除状态。

表 10 总结了 SMART 在第 2 节所述的 3 点挑战上的表现情况。在并发控制方面,SMART 设计的读代理和写合并技术削减了冗余的读写 I/O,这同时也降低了远端锁的冲突程度。在读写放大方面,基数树结构由于其叶子节点仅包含 1 个数据项,因此其读写放大倍数约为 1。在缓存开销方面,基数树每个叶子节点只包含 1 个数据项,因此缓存所有内部节点相当于为每个数据项缓存地址。当装载 6 000 万个数据项时,SMART 的计算端缓存开销为 503 MB[27]。

Table 10 SMART's Solutions to the Challenges of DM Indexes

表 10 SMART 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	读代理与写合并	减少并发冲突
读写放大	基数树	读写放大本身很小
缓存开销	缓存内部节点	缓存开销大

3.2.3 分离式内存上的其他树型索引

除了 B+树和基数树,还有许多工作探索了更多的树型数据结构。dLSM[39] 索引系统实现了针对分离式内存优化的日志结构合并(log-structured merge, LSM)树。它将 LSM 树中的可写内存表(memory table)和冻结内存表(immutable memory table)维护在计算端,而将有序字符串表(sorted string table, SSTable)维护在内存端。这样计算端的用户写操作就可以被吸收并异步、批量地写回内存端。然而,受限于 SSTable 的多层结构特点,dLSM 需要读取多个 SSTable 来寻找目标数据,读性能较低;另外,dLSM 需要利用内存端的 CPU 来进行 SSTable 的压缩合并,容易形成内存端的计算瓶颈。Mariana[40] 索引则实现了针对分离式内存优化的跳表索引。它为叶子节点内的每个数据项维护了单独的锁和校验码,以此实现了细粒度的并发控制。它将叶子节点中数据项的键和值分开存放到 2 个数组中,以实现基于 CPU 单指令多数据流(single instruction multiple data, SIMD)的硬件加速。另外,它设计了精细的计算端跳表缓存用于加速索引遍历。然而,无论是跳表还是 LSM 树,都存在与 B+树一样的读放大问题。表 11 对上述不同数据结构的分离式内存树型索引系统进行了总结对比。

Table 11 Comparison of DM Tree Indexing Systems

表 11 分离式内存树型索引系统对比

系统	结构类型	结构局限
Sherman[17]	B+树	读放大严重
SMART[18]	基数树	缓存开销大
dLSM[39]	LSM 树	读放大严重、压缩合并开销大
Mariana[40]	跳表	读放大严重

3.3 分离式内存混合索引

如表 11 所示,分离式内存树型索引在读放大和缓存效率上存在取舍(trade-off)。具体而言,按照数据项在远端内存的存放方式,可以将树型索引分为 2 个大类:1)离散型索引。该类索引将数据项在内存池上离散地存放(比如基数树),这样用户线程可以直接读取单独的数据项,读放大很小,但是需要为每个数据项缓存地址,缓存开销较大。2)连续型索引。该类索引将数据项在大的叶子节点或大块页面上连续存放(比如 B+树、LSM 树和跳表),这样计算端只需要为每个叶子节点缓存地址,缓存开销很小,但是每次查询数据需要读取整个叶子节点,读放大严重。为此,研究者们开始探索混合的索引结构以优化缓存开销或读放大。本节将介绍 Sphinx, CHIME, Deft 这 3 个代表性的分离式内存混合索引系统。

3.3.1 Sphinx

Sphinx[35] 由上海交通大学提出,是一个混合了基数树索引和哈希索引的分离式内存索引系统。Sphinx 指出分离式内存基数树结构存在缓存效率问题,这导致计算端有限的内存资源可能无法完全缓存所有的内部节点,造成额外的远端树遍历的 RTT 开销。因此,Sphinx 引入了额外的内部节点哈希表结构,这既可以通过并行的哈希表访问加速远端树遍历,又能优化基数树的缓存设计,提高计算端缓存效率。

图 7 展示了 Sphinx 的基本架构。和 SMART 一样,Sphinx 将基数树节点均匀地分布在各个内存节点上。对于每个内存节点上的树节点,Sphinx 维护

了 1 个内部节点的哈希表，将内部节点对应的完整前缀映射到节点的地址和元数据上。内部节点哈希表采用和 RACE 一样的无锁并发控制；而基数树叶子节点则采用和 SMART 类似的基于锁和校验码的并发控制。每个计算节点缓存部分哈希表信息，而不再缓存基数树的内部节点。用户线程根据本地缓存，访问远端哈希表以找到匹配得最深的内部节点，然后再继续访问远端基数树。

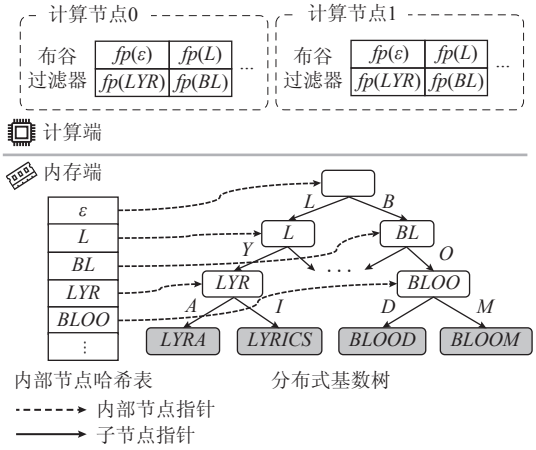


Fig. 7 Basic architecture of Sphinx

图 7 Sphinx 的基本架构

Sphinx 对哈希表与基数树分别沿用现有并发控制技术，解决了大多数的读写一致性问题，但 2 个结构之间仍存在状态同步问题。具体来说，Sphinx 在切换基数树内部节点大小时采用异地更新：当用户线程需要向已满内部节点插入新的指针项时，系统先分配 1 个更大的新节点，再用新节点替换旧节点，并将旧节点从基数树中删除。在哈希表尚未完成同步更新的这段时间内，其他线程仍可能依据旧哈希表定位到已删除的节点，从而产生不一致。为此，Sphinx 在每个内部节点中维护有效位；当发生节点类型切换时，旧节点会被标记为无效。若其他线程通过哈希表访问到无效节点，则执行重试，直到哈希表更新到最新状态。

Sphinx 采用布谷过滤器（cuckoo filter[52]）作为远端内部节点哈希表的缓存，因此这个缓存仅追踪内部节点的存在状态，而不存放内部节点的内容。具体而言，Sphinx 按照布谷过滤器的规则将每个内部节点完整前缀的指纹插入本地哈希表。用户线程首先从长到短依次查询目标数据键的各个前缀是否存在于缓存中；根据查询结果就可以定位匹配得最深的内部节点并直接对最深节点对应的哈希表项发起远端访问。这样，单次操作中的哈希表项读取数量在大多数情况下可由 $O(L)$ 降到 1 条（L 为键长），此时操作可在 3 次网络往返内完成（读取哈希表项、读取内部节点、读取叶子节点）。

该缓存结构的每个过滤器条目使用 10 b 指纹。10 b 指纹在存在性判断中可将误判率控制在较低水平（<1%），同时降低计算端缓存开销。针对误判场景，Sphinx 在后续访问中会结合哈希表项中的 12 b 指纹与内部节点头部的 42 b 完整前缀哈希进行进一步校验；若仍定位到非目标节点，则通过比较公共前缀长度识别误判并以更短前缀重试。该极端情况发生概率在 0.01% 以下。为了进一步针对缓存受限的场景，Sphinx 为存放在布谷过滤器中的每个指纹维护了 1b 的热标记，然后基于 2 次机会页面替换（second-chance page replacement）算法在缓存装满时将冷的缓存项置换出缓存。

基于辅助的哈希表结构，Sphinx 优化了基数树的缓存开销，但额外的数据结构依然引入了少量 RTT 开销。Sphinx 的读写流程如下：1）对于查询操作，用户线程首先查询本地缓存以找到最深的匹配前缀，然后通过 1 次单边 READ 原语读取对应的远端内部节点哈希桶。哈希桶内的每个指针槽中也存放了前缀指纹用于进一步匹配。用户线程从哈希桶内找到指纹匹配的内部节点地址后，再通过 1 次 READ 读取内部节点。如果读取的内部节点就是最深的内部节点，那么用户线程只需要再通过 1 次 READ 读取叶子节点即可；否则需要从这个节点继续遍历基数树。对于范围查询操作，Sphinx 则绕过内部节点哈希表，直接从树根节点开始遍历。2）对于插入或删除操作，用户线程首先通过单边 CAS 原语将数据在远端基数树上插入或删除。如果插入操作触发了基数树的节点分裂，则 Sphinx 会将新的内部节点的前缀指纹插入本地缓存中，并且通过额外的单边 CAS 原语更新远端的内部节点哈希表。3）对于更新操作，用户线程查询到叶子节点后，和 SMART 一样对基数树的叶子节点采用基于锁的就地更新。对于超出叶子节点大小的更长数据项，则和插入操作一样采用异地更新。

表 12 总结了 Sphinx 在第 2 节所述的 3 点挑战上的表现。在并发控制上，Sphinx 沿用现有的哈希表和基数树并发策略，并通过有效位标记来同步 2 种结构的更新状态，避免用户线程读到无效节点。在读写放大上，Sphinx 额外维护了解耦的内部节点哈希表，因此需要额外的 RTT 来维护哈希表内容。在缓存开销上，Sphinx 只需要通过布谷过滤器缓存哈希表信息，因此缓存开销很小。在 Sphinx 的实现中，

Table 12 Sphinx's Solutions to the Challenges of DM Indexes

表 12 Sphinx 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	锁、有效位等	正确同步 2 种数据结构
读写放大	解耦哈希表	额外的哈希表维护开销
缓存开销	布谷过滤器	缓存开销小

其计算端缓存只需要 20 MB[35]。

除了 Sphinx, 还有许多面向缓存优化的混合索引探索。ROLEX[33] 索引用小的机器学习模型替换了 B+树结构中的所有内部节点。这些小模型的内存占用相比 B+树的全部内部节点更小, 可以完全缓存在计算节点中。并且, ROLEX 还利用内存端的计算资源实现了异步的模型重训练过程。Dolphin[32] 索引混合了基数树的内部节点和 B+树的叶子节点。Dolphin 通过增大 B+树叶子节点来减少内部节点数目, 并通过基数树进一步压缩内部节点大小。然而, ROLEX 和 Dolphin 都没有解决 B+树结构的读放大问题。ROLEX 可能还会由于模型误差让读放大问题更加严重。表 13 对分离式内存上面向缓存优化的混合索引系统进行了总结对比。

Table 13 Comparison of Cache-Efficient DM Hybrid Indexing Systems

表 13 缓存优化的分离式内存混合索引系统对比

系统	结构组合	结构局限
Sphinx[35]	基数树与哈希表	哈希表引入额外 RTT
ROLEX[33]	B+树与小模型	读放大严重
Dolphin[32]	基数树与 B+树	读放大严重

3.3.2 CHIME

CHIME[27] 索引由复旦大学提出, 是个基于 B+树内部节点与哈希表叶子节点的混合索引结构。B+树连续的数据存放方式能达到比基数树更高的缓存效率; 并且增大叶子节点大小能进一步减少内部节点数量, 减少缓存开销。但是, B+树的读放大问题严重。为了缓解读放大, 可以将 B+树的叶子节点替换为哈希表, 这样一来: 1)由于数据项依然在叶子节点中连续存放, 该混合索引的缓存开销和 B+树一样低; 2)用户线程不需要读取完整的叶子节点, 只需要读取其中部分的哈希桶, 因此读放大问题也得以缓解。

图 8 展示了 CHIME 的基本架构。CHIME 将树节点均匀地分布在各个内存节点上。内部节点采用的是普通的 B+树结构; 每个叶子节点则被替换为了跳房子哈希表。和其他树型索引一样, 每个计算节点缓存了内部节点以减少树遍历的 RTT。用户线程首先查询本地缓存以定位目标叶子节点, 然后再通过 RDMA 单边原语访问作为叶子节点的跳房子哈希表。

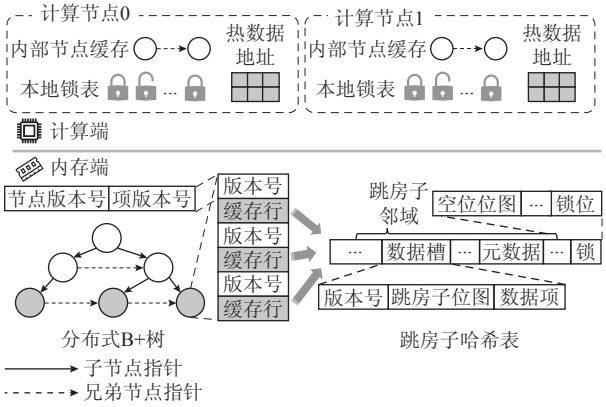


Fig. 8 Basic architecture of CHIME

图 8 CHIME 的基本架构

CHIME 的叶子节点通过锁来处理所有写写冲突。而对于读写冲突, CHIME 则设计了 3 层的校验: 1)对于邻域读操作和 B+树的节点写操作之间的冲突, 采用缓存行版本号来校验。具体而言, CHIME 在每个树节点的头部及其所覆盖的所有缓存行的头部都嵌入了 1 个版本号。节点写操作需要同时更新所有的版本号; 读操作则检查所读取邻域内的版本号是否一致即可。2)对于邻域读操作和单独数据项写操作之间的冲突, CHIME 参考了 Sherman 提出的 2 级版本号思路, 设计了 2 级的缓存行版本号。具体而言, CHIME 在每个数据项的头部也插入缓存行版本号, 并将节点上每个缓存行版本号都分裂为节点版本号和数据项版本号 2 部分。节点写操作只更新所有的节点版本号; 数据项写操作则只更新该数据项所覆盖的所有数据项版本号。这样读操作就可以解耦地校验 2 种粒度的写操作, 即首先检查读取邻域内的节点版本号是否一致, 再检查每个读取的数据项各自的版本号是否一致。这个设计修复了 Sherman 的 2 级首尾版本号的正确性问题, 可以被应用到分离式内存 B+树上。3)对于邻域读操作和哈希表的跳房子操作, 利用跳房子哈希表内部自带的位图来校验数据项的位置是否正确。跳房子哈希表为每个邻域维护了 1 个位图以追踪邻域中每个数据项是否属于该邻域[45]。由于跳房子操作造成的数据移动必然会改变位图信息, 因此读操作可以根据读到的数据重构跳房子位图; 如果重构的位图与读到的位图不一致, 说明数据正在移动, 则读操作需要进行重试。

CHIME 对混合结构中的元数据进行了精细管理。具体而言: 1)跳房子哈希算法需要额外为哈希表维

护 1 个空位位图(vacancy bitmap)，以便插入操作提前确定跳房子的影响范围。为了避免读取位图造成的额外 RTT，CHIME 将空位位图嵌入到了每个叶子节点的 8 B 锁内部，并利用拓展的 CAS 原语[53]实现在获取锁的同时读取锁内部的位图信息。2)B+树的叶子节点需要维护兄弟节点指针(sibling pointer)等元数据，以便读操作在节点分裂期间也能正确定位到目标数据。为了避免读取这些元数据造成的额外 I/O，CHIME 将这些元数据的多个副本均匀放置在叶子节点内部，以保证任意 1 个读邻域都能包含或邻接 1 份元数据。这使得 B+树的元数据和数据总是能被用户线程通过 1 次单边 READ 操作同时获取。

基于上述核心技术，CHIME 设计了精细的读写操作。用户线程无锁地遍历内部节点，在定位叶子节点后：1)对于查询操作，用户线程首先计算目标数据键的哈希值以确定其所在的跳房子邻域。然后通过 1 次单边 READ 原语读取目标邻域，并基于 2 级缓存行版本号、跳房子位图等元数据进行校验。对于范围查询，则采用和 Sherman 一样的 B+树方案。2)对于插入操作，用户线程首先通过 1 次拓展的单边 CAS 操作获取锁和空位位图，再通过 1 次 READ 读取基于空位位图推测出的跳房子范围，并通过 2 级缓存行版本号和跳房子位图进行读写校验。之后，用户线程在读取的数据片段中进行本地跳房子操作。若新数据被成功插入，则通过 1 次单边 WRITE 原语将更新后的叶子节点片段写回到远端内存并将叶子节点解锁。若插入失败，用户线程则对该叶子节点进行 B+树的节点分裂。3)对于更新和删除操作，用户线程和查询操作一样读取数据项，并和插入操作一样对叶子节点上锁。然后，用户线程将更新或清空后的数据项写回远端，并解锁叶子节点。

表 14 总结了 CHIME 在第 2 节所述的 3 点挑战上的表现情况。在并发控制方面，CHIME 采用锁来同步写操作，并设计了 3 级的读写校验机制处理了混合结构带来的复杂并发情形。在读写放大上，CHIME 的基于跳房子哈希表的叶子节点缓解了 B+树的读放大问题，将叶子节点的理论读放大倍数，从 64 倍变为 8 倍。另外，CHIME 和 B+树一样缓存内部节点，二者的缓存开销基本一致。

Table 14 CHIME's Solutions to the Challenges of DM Indexes

表 14 CHIME 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	3 级读写校验	实现正确的乐观同步
读写放大	跳房子哈希叶子	缓解 B+树的读放大
缓存开销	缓存内部节点	缓存开销较小

3.3.3 Deft

Deft[20]索引由清华大学提出，它与 CHIME 类似，也是采用 B+树内部节点与哈希叶子节点的混合索引结构。不过，Deft 在具体的结构设计与并发控制算法上有所不同。

图 9 展示了 Deft 的整体结构。Deft 将树节点分布在内存端。与传统 B+树不同，Deft 把内部节点组织为分段结构，把叶子节点组织为哈希结构。内部节点按键范围划分为多个子段(1 KB 节点默认划分为 4 段)，子段之间保持有序关系，子段内的数据采用无序存放；叶子节点则采用类似 RACE 哈希表的设计，由多个主桶和溢出桶组成，2 个主桶共享 1 个位于中间的溢出桶。在并发控制上，Deft 则与 Sherman 类似，将锁与树节点解耦，通过节点地址哈希到锁表。

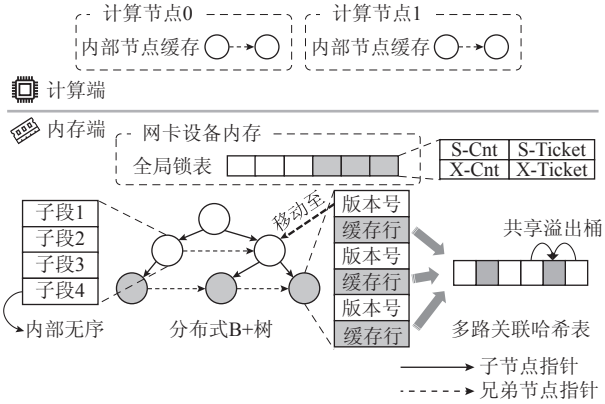


Fig. 9 Basic architecture of Deft

图 9 Deft 的基本架构

Deft 通过分段内部节点与哈希叶子节点同时缓解内部节点层与叶子节点层的读放大问题。在内部节点层中，查询先按键范围定位目标子段，再执行子段级读取，读粒度从整节点降为 1 个子段；插入时由于子段内部的数据无序，因此不需要执行传统有序节点的数据整体搬移。若子段写满，Deft 执行相邻子段的两两合并以形成更大范围的子段，这一过程仅更新粒度信息与父节点中的元数据；若内部节点整体写满，Deft 做整体节点分裂。该设计不引入额外子段指针，不降低节点扇出，也不增加树高。在叶子节点层中，Deft 把点查询的读取范围缩小到 RACE 哈希的 1 对连续哈希桶；当主桶与溢出桶同时装满数据时才进行叶子节点的分裂，其过程与哈希表扩容相同。

在写写冲突处理上, Deft 采用共享-互斥锁。对于同个叶子节点上的插入和更新操作, Deft 以共享方式推进, 而对于 SMO 操作(如节点分裂)则以独占方式执行, 二者互斥, 这样能有效减少单个节点内的并发冲突。每个锁是 64 b 结构, 包含 S-Cnt, S-Ticket, X-Cnt, X-Ticket 这 4 个 16 b 字段, Deft 通过拓展的单边 FAA 操作原子地为各个字段增加值。具体而言, 当用户线程需要获取共享锁时, 它通过 1 次拓展的 FAA 操作将 S-Ticket 值加 1 并获取 X-Ticket 和 X-Cnt 值; 若 X-Ticket 和 X-Cnt 相等则说明互斥锁没有被占用, 则共享锁获取成功; 否则, 用户线程需要通过 READ 操作轮询 X-Cnt 直到检测到互斥锁被释放。当用户线程需要获取互斥锁时, 它通过拓展的 FAA 操作将 X-Ticket 值加 1 并获取所有的 4 个字段值; 若 S-Ticket 和 S-Cnt 相等且 X-Ticket 和 X-Cnt 相等, 则说明共享锁和互斥锁均未被占用, 此时互斥锁获取成功; 否则, 用户线程需要轮询 S-Cnt 和 X-Cnt 直到检测到 2 种锁都被释放。对于锁的释放, 用户线程通过增加 S-Cnt 来释放共享锁, 通过增加 X-Cnt 来释放互斥锁。

Deft 通过结合缓存行版本号与查询路径上的父子节点版本号保证正确的无锁读操作。具体而言, Deft 在节点侧维护缓存行版本号, 即在节点所覆盖的所有缓存行的头部都嵌入 1 个版本号, 然后在父节点中指向该节点的指针内也嵌入 1 个相同的版本号。当用户线程没有触发 SMO 时, 节点内的缓存行版本号足以用于维护数据读写一致性。而当用户线程触发了 SMO 操作时, 它首先在更新节点数据的同时更新节点内的所有缓存行版本号, 然后在修改父节点中指针的同时也一并将其中为子节点所维护的版本号进行更新; 读操作在搜索路径上可以先后读取到父节点内的指针版本号与子节点的缓存行版本号, 若 2 种版本号不一致则说明 SMO 操作尚未完成, 此时读操作需要进行重试。

基于上述核心技术, Deft 设计了精细的读写操作: 1)对于查询操作, 用户线程首先遍历分段内部节点。对于路径上的每个内部节点先依据键范围与粒度信息定位目标子段, 然后通过单边 READ 操作读取目标子段。到达叶子节点后, 用户线程根据目标数据键的哈希值读取目标哈希桶与溢出桶。整个遍历过程会基于路径上的父子节点版本号进行读写一致性校验, 不一致则重试。对于范围查询, Deft 则采用和 B+树一样的方案。2)对于插入和更新操作, 用户线程先按共享模式获取共享-互斥锁; 在叶子节点中定位目标槽位后, Deft 对小的数据项直接用拓展的 CAS 操作原子写入, 对于大的数据项则通过额外的锁字段来实现数据原子写入; 若节点触发 SMO 操作, Deft 则将在进一步获取互斥锁后才执行 SMO 操作。3)对于删除操作, Deft 直接获取互斥锁以避免并发的删除与插入操作引入重复键。

表 15 总结了 Deft 在第 2 节所述的 3 点挑战上的表现情况。在并发控制方面, Deft 采用共享-互斥锁来同步不同粒度的写操作, 并设计了查询路径上的父子节点版本号来保证正确的无锁读操作。在读写放大上, Deft 通过分段的内部节点与基于哈希表的叶子节点缓解了 B+树的读放大问题, 将内部节点与叶子节点的理论读放大倍数分别变为了 15 和 8。在缓存开销上, Deft 与 B+树一样缓存内部节点, 缓存开销与 B+树基本一致。

Table 15 Deft's Solutions to the Challenges of DM Indexes

表 15 Deft 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	共享-互斥锁、父子版本号	减少冲突粒度
读写放大	分段内部节点、哈希叶子	缓解读放大
缓存开销	缓存内部节点	缓存开销较小

3.4 基于 CXL 互联技术的分离式内存索引

近年来, 随着 CXL 技术的发展, 研究者开始探索如何在基于 CXL 互联的分离式内存系统上构建高效索引结构。与 RDMA 网络相比, CXL 通常具有更高的带宽和更低的时延, 因此被认为是分离式内存系统中颇具前景的互联方案。CHash[16] 是个具有代表性的基于 CXL 互联技术的分离式内存哈希索引系统。

CHash 索引由阿里云提出, 它指出, 尽管 CXL 为分离式内存系统提供了比传统网络互联更高的带宽和更强的容量扩展能力, 但真实 CXL 内存设备在访问时延、带宽利用率以及内存分配开销等方面仍与本地 DRAM 存在明显差距。若直接沿用面向 DRAM 或持久化内存设计的哈希索引方案, 往往难以充分发挥 CXL 设备的性能。为此, CHash 提出了面向 DRAM 和 CXL 混合存储的哈希结构: 将哈希表目录和高频访问的元数据维护在 DRAM 中, 而将主要的哈希桶存放在 CXL 内存中, 以降低查询与更新路径上的高代价 CXL 访问。

图 10 展示了 CHash 的整体结构。CHash 采用多路关联哈希的组织方式, 在每个哈希桶中连续存放多个数据项。在此基础上, 它将多个桶进一步组织为段(segment), 并在目录项中保存对应段的指针及相关元数据。与传统方案不同, CHash 在每个目录

项中额外维护了 1 个位于 DRAM 中的暂存桶(stage bucket)，用于暂存新插入的数据项；当暂存桶写满后，CHash 再将其中的数据批量迁移到 CXL 内存中的目标桶中。CXL 内存中的哈希桶与暂存桶采用相同布局，因此可以直接完成迁移，而无需额外格式转换。与此同时，CHash 在每个目录项中维护段级与桶级元数据，包括段指针、段位图、锁、版本号、布隆过滤器和指纹数组等内容，从而为后续查找与扩容操作提供支持。

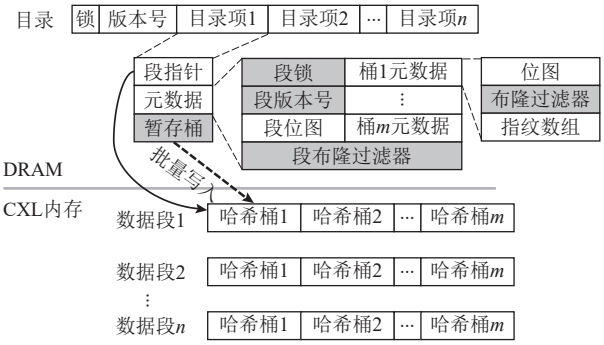


Fig. 10 Basic architecture of CHash

图 10 CHash 的基本架构

针对 CXL 内存读写代价较高的特点，CHash 设计了多项加速技术。首先，暂存桶机制利用 DRAM 对写入请求进行缓冲和聚合，避免将大量细粒度写操作直接发送到 CXL 内存，从而提升写带宽利用率。其次，CHash 采用 2 级过滤机制：目录项中维护段级布隆过滤器，用于快速判断目标段中是否可能包含目标键；同时维护桶级布隆过滤器，进一步缩小可能命中的桶范围，减少无效桶访问。再次，CHash 为每个哈希桶维护指纹数组，在真正执行键比较之前，先对 DRAM 中的指纹数组进行匹配，仅对指纹匹配的槽位进行精确比较，以降低哈希桶内数据比较带来的额外访存开销。此外，CHash 还利用单次哈希计算结果同时生成段索引、过滤器和指纹信息，从而减少重复哈希计算带来的开销。

在哈希扩容方面，CHash 采用类似可扩展哈希的目录扩展机制。在分配出新的数据段后，CHash 重新扫描旧段中的数据项，并对需要重新哈希的数据进行迁移。与传统方式不同，CHash 依据新的目录映射，将这些数据项逐个重新插入新目录项对应的暂存桶，而不是直接在 CXL 侧执行细粒度原地迁移。这样一方面可以在重分布过程中顺带完成哈希桶的空位压缩，缓解删除带来的碎片问题；另一方面也能够同时重建布隆过滤器与指纹元数据，降低误判率。

在并发控制上，CHash 采用乐观并发控制与段级锁相结合的方案。写线程通过 CAS 获取目标段的锁，在临界区内完成插入、迁移或扩容等操作，并在结束后递增版本号；读线程则不会加锁，只在操作结束时校验版本号，若版本相较于操作开始前发生变化，则整体重试。

基于上述核心设计，CHash 对各种读写操作进行了细致的设计：1)对于查询操作，用户线程首先根据哈希值定位目标目录项，并优先检查段级布隆过滤器，以判断目标键是否可能位于对应段中；若段级过滤通过，则进一步检查各个桶的桶级布隆过滤器和指纹数组，仅对可能命中的桶中指纹匹配的数据进行读取。除 CXL 内存中的哈希桶外，查询还需要检查目录项中的暂存桶，以避免遗漏尚未迁移到 CXL 侧的新插入数据。在整个查询完成后，线程还需校验目录项版本号，若版本发生变化则重新执行查询。2)对于插入和更新操作，用户线程首先获取目标段的锁，并执行与查询类似的流程；若目标键已存在，则直接写入新值。若暂存桶未满，线程将新的数据项直接写入暂存桶中的空槽，并同步更新相应的段级过滤器、桶级过滤器和指纹元数据；若暂存桶已满，则用户线程先尝试将暂存桶中的数据批量迁移到目标段中的空桶；若目标段也已满，则触发段分裂与目录扩展，待扩容完成后重新执行插入。3)对于删除操作，用户线程首先按照查询路径定位目标键所在的槽位；若成功找到目标项，则将对应槽位的有效位清零，使该槽位可供后续插入复用。为了降低删除开销，CHash 在删除时不会立即更新相关过滤器与指纹结构，而是在后续扩容和重分布过程中统一地重建这些元数据。

表 16 总结了 CHash 在第 2 节所述的 3 点挑战上的表现情况。在并发控制方面，CHash 通过段粒度的锁和版本号实现了乐观并发控制。在读写放大上，CHash 通过 DRAM 存放的 2 级的布隆过滤器和指纹数组来尽可能减少对 CXL 内存的读取。在缓存开销上，CHash 除了缓存目录，还需要缓存暂存桶和元数据，其每个目录项的缓存大小约 2 KB，因此整体缓存开销仅为数百 KB 量级。

Table 16 CHash's Solutions to the Challenges of DM Indexes

表 16 CHash 对分离式内存索引挑战的解决情况

维度	技术点	效果或局限
并发控制	段粒度锁、版本号	正确实现乐观同步
读写放大	2 级布隆过滤器、指纹数组	减少 CXL 读开销
缓存开销	缓存目录、暂存桶和元数据	缓存开销小

4 总结与展望

本文首先从分离式内存的架构特点和RDMA网卡的性能瓶颈出发，展开分析了构建高性能的分离式内存索引系统面临的挑战，然后对现有研究工作进行了详细的分类、剖析与对比。现有研究工作针对分离式内存架构中各层的资源约束，利用RDMA网卡的单边远程访存能力，对各种常见的索引数据结构进行了加速，包括哈希索引、树型索引以及混合索引等，能够显著提高分离式内存上数据的索引效率。

本文综述的研究工作大多从数据结构设计方面来提升分离式内存索引系统，未来的研究工作可以从更多维度进行深入探索，如节点协作、容错机制以及新软硬件适配等，以进一步推动分离式内存索引系统的研究与发展。

4.1 节点协作

现有的分离式内存索引系统大多让每个计算节点孤立地访问远端内存池，难以充分利用网络资源并进行全局性能优化。一方面，计算节点独立的访存模式使得内存端网卡的带宽或消息处理速率成为性能瓶颈，而计算端的网络资源则未被充分利用。通过引入计算节点间的协作有望进一步缓解内存端网络瓶颈，提升分离式内存索引系统的吞吐率。另一方面，计算节点独立的访存模式使得现有的部分优化技术实现不彻底。例如，SMART提出的读代理和写合并技术仅消除了单个计算节点内的冗余I/O，而没有实现全局的优化；DINOMO和DEX采用的归属划分方案仅对数据归属进行了静态划分，而没有实现计算节点间的负载均衡。

事实上，已经有一些研究工作正在探讨通过计算节点协作来优化远端内存系统的性能。清华大学基于MCS(Mellor-Crummey and Scott)锁机制成功地将Sherman中的锁移交机制拓展到了整个计算端[54-55]，并获得了较大性能提升；中国科学技术大学提出的DMTree[56]将部分索引操作(如锁操作、数据项定位操作)卸载到其他计算节点，充分利用了计算侧的RDMA资源。

4.2 容错机制

现有的分离式内存索引系统大多关注于性能优化，而没有考虑容错问题。在分离式内存架构下，计算节点和内存节点可以独立发生故障，这产生了复杂的故障情形。首先，由于所有索引上的数据和元数据都由计算节点使用RDMA单边操作直接修改，在计算节点单独发生故障时，数据和元数据会被部分修改，产生数据污染的情况。同时，其他处于正常状态的计算节点可以使用RDMA单边操作访问到这些被污染的数据，导致系统出现正确性问题。其次，分离式内存上的索引通常采用复杂的并发控制机制。若计算节点在持有远端锁时故障，索引系统将会产生死锁问题。最后，内存节点的故障会导致数据丢失，同样也会影响到计算节点的请求执行，导致分布式索引的可用性降低。另外，当引入计算节点间或计算与内存节点间的协同设计时，索引系统的故障域会进一步扩大。因此，高效的容错机制对于实现高性能、高可用的分离式内存索引系统至关重要。

4.3 新软硬件适配

随着CXL技术的发展，越来越多的研究开始探索将CXL互联技术应用到分离式内存架构中，以享受其更高的带宽和更低的时延[15-16]。如何将现有的索引设计高效地迁移到CXL设备上是个值得探索的问题。另外，CXL技术在拓展性方面仍存在局限性，难以适用于机柜间互联的场景。因此，基于CXL与RDMA网络的异构内存池索引系统也将是一个极具实践意义的研究方向[57]。

另外，在软件方面，随着RDMA驱动的升级与换代，其提供的用户原语逐渐趋于保守。然而，许多最新的分离式内存索引系统仍然使用拓展的RDMA原语来达到额外的索引加速效果。例如，RCuckoo[26]和Sherman[17]采用了基于掩码的CAS原语以压缩设备内存内锁表的空间开销；Deft[20]采用了基于掩码的FAA原语以实现互斥-共享锁以及拓展的CAS原语以支持32 B的原子操作；CHIME[27]采用了基于掩码的CAS原语以优化空位位图的读I/O等。这些实验性的拓展原语有许多在最新RDMA驱动中已经不再被支持[58]。因此，在未来的分离式内存索引设计中，研究者们应该尽可能兼顾硬件与软件通用性，以推动分离式内存索引系统被更广泛地应用在各种数据中心中。

作者贡献声明：罗旭川负责文献的搜集整理、论文整体架构设计和论文主要内容的撰写；沈家诚负责论文主要内容的修改和部分内容的撰写；周扬帆和王新负责提出指导意见并修改论文。

参 考 文 献

[1] Li Huaicheng, Berger D, Hsu L, et al. Pond: CXL-based memory

pooling systems for cloud platforms[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23). New York: ACM, 2023: 574−587

[2] Liu Qixiao, Yu Zhibin. The elasticity and plasticity in semi-containerized co-locating cloud workload: A view from Alibaba trace[C]//Proc of the 9th ACM Symp on Cloud Computing (SoCC'18). New York: ACM, 2018: 347−360

[3] Tirmazi M, Barker A, Deng Nan, et al. Borg: The next generation[C]//Proc of the 15th European Conf on Computer Systems (EuroSys'20). New York: ACM, 2020: 30: 1−30: 14

[4] Gao P, Narayan A, Karandikar S, et al. Network requirements for resource disaggregation[C]//Proc of the 12th USENIX Symp on Operating Systems Design and Implementation (OSDI'16). Berkeley, CA: USENIX Association, 2016: 249−264

[5] Shan Yizhou, Huang Yutong, Chen Yilun, et al. LegoOS: A disseminated, distributed OS for hardware resource disaggregation[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI'18). Berkeley, CA: USENIX Association, 2018: 69−87

[6] Cao Wei, Zhang Yingqiang, Yang Xinjun, et al. PolarDB Serverless: A cloud native database for disaggregated data centers[C]//Proc of the 47th Int Conf on Management of Data (SIGMOD'21). New York: ACM, 2021: 2477−2489

[7] InfiniBand Trade Association. Enabling the modern data center--RDMA for the enterprise[EB/OL]. [2025-10-10]. https://www.infinibandta.org

[8] CXL Consortium. CXL 3.2 specification now available[EB/OL]. [2025-10-10]. https://www.computeexpresslink.org

[9] Shu Jiwu, Chen Youmin, Wang Qing, et al. Progress on storage systems for disaggregated data centers[J]. SCIENTIA SINICA Informationis, 2023, 53(8): 1503−1528 (in Chinese)
(舒继武, 陈游旻, 汪庆, 等. 面向分离式数据中心的存储系统研究进展[J]. 中国科学: 信息科学, 2023, 53(8): 1503−1528)

[10] Huang Mandi, Li Tao, Yang Hui, et al. Survey on Ethernet RDMA network interface card[J]. Journal of Computer Research and Development, 2025, 62(5): 1262−1289 (in Chinese)
(黄曼蒂, 李韬, 杨惠, 等. 以太网 RDMA 网卡综述[J]. 计算机研究与发展, 2025, 62(5): 1262−1289)

[11] Kalia A, Kaminsky M, Andersen D. Design guidelines for high performance RDMA systems[C]//Proc of the 2016 USENIX Annual Technical Conf (ATC'16). Berkeley, CA: USENIX Association, 2016: 437−450

[12] NVIDIA Corporation. NVIDIA ConnectX-7 400G adapters[EB/OL]. [2025-10-10]. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/connectx-7-datasheet

[13] Su Maomeng, Zhang Mingxing, Chen Kang, et al. RFP: When RPC is faster than server-bypass with RDMA[C]//Proc of the 12th European Conf on Computer Systems (EuroSys'17). New York: ACM, 2017: 1−15

[14] Ren Feng, Zhang Mingxing, Chen Kang, et al. Scaling up memory disaggregated applications with SMART[C]//Proc of the 29th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24). New York: ACM, 2024: 351−367

[15] Yang Xinjun, Zhang Yingqiang, Chen Hao, et al. Unlocking the potential of CXL for disaggregated memory in cloud-native databases[C]//Proc of the 51st Int Conf on Management of Data (SIGMOD'25). New York: ACM, 2025: 689−702

[16] Lu Mengting, Liu Gaocong, Wang Kun, et al. CHash: A high cost-performance Hash design for CXL-based disaggregated memory system[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2025, 9(1): 5:1−5:22

[17] Wang Qing, Lu Youyou, Shu Jiwu. Sherman: A write-optimized distributed B+ tree index on disaggregated memory[C]//Proc of the 48th Int Conf on Management of Data (SIGMOD'22). New York: ACM, 2022: 1033−1048

[18] Luo Xuchuan, Zuo Pengfei, Shen Jiacheng, et al. SMART: A high-performance adaptive radix tree for disaggregated memory[C]//Proc of the 17th USENIX Symp on Operating Systems Design and Implementation (OSDI'23). Berkeley, CA: USENIX Association, 2023: 553−571

[19] An Hang, Wang Fang, Feng Dan, et al. Marlin: A concurrent and write-optimized B+-tree index on disaggregated memory[C]//Proc of the 52nd Int Conf on Parallel Processing (ICPP'23). New York: ACM, 2023: 695−704

[20] Wang Jing, Wang Qing, Zhang Yuhao, et al. Deft: A scalable tree index for disaggregated memory[C]//Proc of the 20th European Conf on Computer Systems (EuroSys'25). New York: ACM, 2025: 886−901

[21] Lu Baotong, Huang Kaisong, Liang Chieh-Jan M, et al. DEX: Scalable range indexing on disaggregated memory[J]. Proceedings of the VLDB Endowment, 2024, 17(10): 2603−2616

[22] Lee S, Ponnapalli S, Singhal S, et al. DINOMO: An elastic, scalable, high-performance key-value store for disaggregated persistent memory[J]. Proceedings of the VLDB Endowment, 2022, 15(13): 4023−4037

[23] Yang Juncheng, Yue Yao, Rashmi K. A large scale analysis of hundreds of in-memory cache clusters at Twitter[C]//Proc of the 14th USENIX Symp on Operating Systems Design and Implementation (OSDI'20). Berkeley, CA: USENIX Association, 2020: 191−208

[24] Atikoglu B, Xu Yuehai, Frachtenberg E, et al. Workload analysis of a large-scale key-value store[C]//Proc of the 12th ACM Joint Int Conf on Measurement and Modeling of Computer Systems (SIGMETRICS'12). New York: ACM, 2012: 53−64

[25] Zuo Pengfei, Sun Jiazhao, Yang Liu, et al. One-sided RDMA-conscious extendible hashing for disaggregated memory[C]//Proc of the 2021 USENIX Annual Technical Conf (ATC'21). Berkeley, CA: USENIX Association, 2021: 15−29

[26] Grant S, Snoeren A. Cuckoo for clients: Disaggregated cuckoo hashing[C]//Proc of the 2025 USENIX Annual Technical Conf (ATC'25). Berkeley, CA: USENIX Association, 2025: 959−972

[27] Luo Xuchuan, Shen Jiacheng, Zuo Pengfei, et al. CHIME: A cache-efficient and high-performance hybrid index on disaggregated memory[C]//Proc of the 30th ACM SIGOPS Symp on Operating Systems Principles (SOSP'24). New York: ACM, 2024: 110−126

[28] Dragojevic A, Narayanan D, Hodson O, et al. FaRM: Fast remote memory[C]//Proc of the 11th USENIX Symp on Networked Systems Design and Implementation (NSDI'14). Berkeley, CA: USENIX Association, 2014: 401−414

[29] NVIDIA Corporation. RDMA aware networks programming user manual v1.7[EB/OL]. [2025-10-10]. https://docs.nvidia.com/networking/display/rdmaawareprogrammingv17

[30] Ziegler T, Nelson-Slivon J, Leis V, et al. Design guidelines for correct,

efficient, and scalable synchronization using one-sided RDMA[J]. Proceedings of the ACM on Management of Data, 2023, 1(2): 131:1−131:26

[31] Min Xinhao, Lu Kai, Liu Pengyu, et al. SepHash: A write-optimized Hash index on disaggregated memory via separate segment structure[J]. Proceedings of the VLDB Endowment, 2024, 17(5): 1091−1104

[32] An Hang, Wang Fang, Feng Dan, et al. Dolphin: A resource-efficient hybrid index on disaggregated memory[C/OL]//Proc of the 50th Int Conf on Massive Storage Systems and Technology (MSST'24). Piscataway, NJ: IEEE, 2024 [2025-10-10]. https://www.msstconference.org/MSST-history/2024/Papers/msst24-8.2.pdf

[33] Li Pengfei, Hua Yu, Zuo Pengfei, et al. ROLEX: A scalable RDMA-oriented learned key-value store for disaggregated memory systems[C]//Proc of the 21st USENIX Conf on File and Storage Technologies (FAST'23). Berkeley, CA: USENIX Association, 2023: 99−114

[34] Wei Xinda, Chen Rong, Chen Haibo. Fast RDMA-based ordered key-value store using remote learned cache[C]//Proc of the 14th USENIX Symp on Operating Systems Design and Implementation (OSDI'20). Berkeley, CA: USENIX Association, 2020: 117−135

[35] Li Jingxiang, Zheng Shenan, Zhang Bowen, et al. Sphinx: A high-performance hybrid index for disaggregated memory with succinct filter cache[C]//Proc of the 62nd ACM/IEEE Design Automation Conf (DAC'25). Piscataway, NJ: IEEE, 2025: 1−7

[36] Liu Yi, Xie Minghao, Shi Shouqian, et al. Outback: Fast and communication-efficient index for key-value store on disaggregated memory[J]. Proceedings of the VLDB Endowment, 2024, 18(2): 335−348

[37] Ziegler T, Vani S, Binnig C, et al. Designing distributed tree-based index structures for fast RDMA-capable networks[C]//Proc of the 45th Int Conf on Management of Data (SIGMOD'19). New York: ACM, 2019: 741−758

[38] Wan Tianyu, Gong Shijia, Hu Yangyang, et al. GECKO: A write-optimized adaptive radix tree for disaggregated memory[C]//Proc of the 31st European Conf on Parallel and Distributed Processing (Euro-Par'25). Berlin: Springer, 2025: 239−253

[39] Wang Ruihong, Wang Jianguo, Kadam P, et al. dLSM: An LSM-based index for memory disaggregation[C]//Proc of the 39th IEEE Int Conf on Data Engineering (ICDE'23). Piscataway, NJ: IEEE, 2023: 2835−2849

[40] Wei Xing, Wang Ke, Han Yinjun, et al. Mariana: Exploring native skiplist index design for disaggregated memory[J]. IEEE Transactions on Parallel and Distributed Systems, 2025, 36(10): 2137−2151

[41] Aguilera M, Keeton K, Novakovic S, et al. Designing far memory data structures: Think outside the box[C]//Proc of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19). New York: ACM, 2019: 120−126

[42] Shen Jiacheng, Zuo Pengfei, Luo Xuchuan, et al. FUSEE: A fully memory-disaggregated key-value store[C]//Proc of the 21st USENIX Conf on File and Storage Technologies (FAST'23). Berkeley, CA: USENIX Association, 2023: 81−98

[43] Shen Jiacheng, Zuo Pengfei, Luo Xuchuan, et al. Ditto: An elastic and adaptive memory-disaggregated caching system[C]//Proc of the 29th ACM SIGOPS Symp on Operating Systems Principles (SOSP'23). New York: ACM, 2023: 675−691

[44] Pagh R, Rodler F. Cuckoo hashing[J]. Journal of Algorithms, 2004, 51(2): 122−144

[45] Herlihy M, Shavit N, Tzafrir M. Hopscotch hashing[C]//Proc of the 22nd Int Symp on Distributed Computing (DISC'08). Berlin: Springer, 2008: 350−364

[46] Mitzenmacher M. The power of two choices in randomized load balancing[J]. IEEE Transactions on Parallel and Distributed Systems, 2001, 12(10): 1094−1104

[47] Zuo Pengfei, Hua Yu, Wu Jie. Write-optimized and high-performance hashing index scheme for persistent memory[C]//Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI'18). Berkeley, CA: USENIX Association, 2018: 461−476

[48] Shi Shouqian, Qian Chen. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2020, 4(2): 22: 1−22: 32

[49] Leis V, Kemper A, Neumann T. The adaptive radix tree: ARTful indexing for main-memory databases[C]//Proc of the 29th IEEE Int Conf on Data Engineering (ICDE'13). Piscataway, NJ: IEEE, 2013: 38−49

[50] Leis V, Scheibner F, Neumann T. The ART of practical synchronization[C]//Proc of the 12th Int Workshop on Data Management on New Hardware (DaMoN'16). New York: ACM, 2016: 3: 1−3: 8

[51] Intel Corporation. Write combining memory implementation guidelines[EB/OL]. [2025-10-10]. https://download.intel.com/design/PentiumII/applnots/24442201.pdf

[52] Fan Bin, Andersen D, Kaminsky M, et al. Cuckoo filter: Practically better than Bloom[C]//Proc of the 10th ACM Int Conf on Emerging Networking Experiments and Technologies (CoNEXT'14). New York: ACM, 2014: 75−88

[53] NVIDIA Corporation. RDMA aware networks programming user manual[EB/OL]. [2026-10-23]. https://docs.nvidia.com/rdma-aware-networks-programming-user-manual-1-7.pdf

[54] Gao Jian, Wang Qing, Shu Jiwu. ShiftLock: Mitigate one-sided RDMA lock contention via handover[C]//Proc of the 23rd USENIX Conf on File and Storage Technologies (FAST'25). Berkeley, CA: USENIX Association, 2025: 355−372

[55] Gao Jian, Shu Jiwu. An efficient distributed lock protocol in remote direct memory access network[J]. Journal of Computer Research and Development, 2026, 63(3): 585−596 (in Chinese)
(高健, 舒继武. 一种远程直接内存访问网络中的高效分布式锁协议[J]. 计算机研究与发展, 2026, 63(3): 585−596)

[56] Wei Guoli, Li Yongkun, Song Haoze, et al. DMTree: Towards efficient tree indexing on disaggregated memory via compute-side collaborative design[C]//Proc of the 24th USENIX Conf on File and Storage Technologies (FAST'26). Berkeley, CA: USENIX Association, 2026: 149−165

[57] Zhang Xu, Liu Ke, Hui Yuan, et al. DRack: A CXL-disaggregated rack architecture to boost inter-rack communication[C]//Proc of the 2025 USENIX Annual Technical Conf (ATC'25). Berkeley, CA: USENIX Association, 2025: 1261−1279

[58] NVIDIA Corporation. MLNX_OFED Software Release Notes[EB/OL]. [2026-10-14]. https://networking-docs.nvidia.com/mlnxofedsw-um/24.10-5.1.6.1lts/release-notes



Luo Xuchuan, born in 1999. PhD. Student member of CCF. His main research interest includes disaggregated memory and indexing systems.

罗旭川, 1999 年生。博士。CCF 学生会员。主要研究方向为分离式内存与索引系统。



Shen Jiacheng, born in 1997. PhD, assistant professor. Member of CCF. His main research interests include distributed storage systems and machine learning systems.

沈家诚, 1997 年生。博士，助理教授。CCF 会员。主要研究方向为分布式存储系统、机器学习系统。



Zhou Yangfan, born in 1979. PhD, professor, PhD supervisor. Member of CCF. His main research interests include system software and reliability engineering.

周扬帆, 1979 年生。博士，教授，博士生导师。CCF 会员。主要研究方向为系统软件、可靠性工程。



Wang Xin, born in 1973. PhD, professor, PhD supervisor. Distinguished member of CCF. His main research interest includes distributed storage systems.

王　新, 1973 年生。博士，教授，博士生导师。CCF 杰出会员。主要研究方向为分布式存储系统。