计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220502
60(3): 509−524, 2023

ZB$^+$-tree: 一种 ZNS SSD 感知的新型索引结构

刘 扬 金培权
(中国科学技术大学计算机科学与技术学院 合肥 230026)
(中国科学院电磁空间信息重点实验室 合肥 230026)
(liuyang32@mail.ustc.edu.cn)

ZB$^+$-tree: A Novel ZNS SSD-Aware Index Structure

Liu Yang and Jin Peiquan
(*School of Computer Science and Technology* , *University of Science and Technology of China* , *Hefei* 230026)
(*Key Laboratory of Electromagnetic Space Information* , *Chinese Academy of Sciences* , *Hefei* 230026)

Abstract ZNS SSD is a newly emerging solid state drive (SSD). It supports zone-based storage and access of data in SSD. Compared with traditional SSDs, ZNS SSD can effectively improve the read-write throughput of SSD, reduce write amplification and over-provisioning in SSD. However, ZNS SSD requires that data must be written to zones sequentially, and tasks such as space allocation and garbage collection need to be controlled by users. These characteristics of ZNS SSD pose new challenges to many components in traditional database systems, such as storage management, indexing, and buffer management. This paper mainly studies how to adapt the traditional B$^+$-tree index structure to ZNS SSD. We propose a new ZNS SSD-aware index structure called ZB$^+$-tree (ZNS-aware B$^+$-tree), which is the first ZNS SSD-aware index so far. ZB$^+$-tree takes B$^+$-tree as the base structure, and utilizes the two kinds of zones inside ZNS SSD, namely conventional zones supporting a few random writes and sequential zones only supporting sequential writes. In particular, it uses conventional zones to absorb random writes to ZNS SSD and stores the index nodes in conventional and sequential zones separately. We design different node structures for the nodes in the two zones. By using different types of nodes on the two zones, ZB$^+$-tree can absorb random writes on the index and ensure the sequential write requirements on sequential zones. We conduct experiments by simulating ZNS SSD using null_blk and libzbd. Also, we modify the existing copy-on-write (CoW) B$^+$-tree as the competitor. The results show that ZB$^+$-tree outperforms CoW B$^+$-tree in various metrics including running time and space efficiency.

Key words ZNS SSD; B$^+$-tree; database index; CoW B$^+$-tree; zoned storage

摘 要 ZNS SSD 是近年来提出的一种新型固态硬盘(solid state drive, SSD),它以分区(Zone)的方式管理和存取 SSD 内的数据.相比于传统 SSD, ZNS SSD 可以有效提升 SSD 的读写吞吐,降低写放大,减少 SSD 的预留空间.但是, ZNS SSD 要求 Zone 内必须采用顺序写模式,并且 Zone 上的空间分配、垃圾回收等任务都需要用户自行控制.ZNS SSD 的这些特性对于传统数据库系统的存储管理、索引、缓存等技术均提出了新的挑战.针对如何使传统的 B$^+$-tree 索引结构适配 ZNS SSD 的问题,提出了一种 ZNS SSD 感知的新型索引结构——ZB$^+$-tree (ZNS-aware B$^+$-tree).ZB$^+$-tree 是目前已知的首个 ZNS SSD 感知的索引,它以 B$^+$-tree 为基础,利用 ZNS SSD 内部支持少量随机写的常规 Zone(conventional zone, Cov-Zone)和只支持顺序写的顺序 Zone(sequential zone, Seq-Zone),通过常规 Zone 来吸收对 ZNS SSD 的随机写操作.ZB$^+$-tree 将索

收稿日期: 2022-06-11; **修回日期**: 2022-11-04
基金项目: 国家自然科学基金项目(62072419)
This work was supported by the National Natural Science Foundation of China (62072419).
通信作者: 金培权 (jpq@ustc.edu.cn)

引节点分散存储在常规 Zone 和顺序 Zone 中，并为 2 种 Zone 内的节点分别设计了节点结构，使 ZB$^+$-tree 不仅能够吸收对索引的随机写操作，而且又可以保证顺序 Zone 内的顺序写要求.在实验中利用 null_blk 和 libzbd 模拟 ZNS SSD 设备，并将现有的 CoW B$^+$-tree 修改后作为对比索引.结果表明，ZB$^+$-tree 在运行时间、空间利用率等多个指标上均优于 CoW B$^+$-tree.

关键词 ZNS SSD; B$^+$树; 数据库索引; CoW B$^+$-tree; 分区存储

中图法分类号 TP311.13

ZNS SSD (Zoned Namespaces SSD)[1-7] 是 2019 年由西部数据公司和三星公司推出的新一代固态硬盘(solid state drive, SSD)，目前受到了工业界和学术界的广泛关注. 由于基于闪存的 SSD只有在块被完全擦除以后才能重写，传统的 SSD 通过使用闪存转换层(flash translation layer, FTL)[8] 来适应这一特性，但由于闪存块存在物理限制(擦除操作以块大小进行，而读写操作以页大小进行)，因此经常需要进行垃圾回收(garbage collection, GC)[5,9]，同时也带来了预留空间(over provisioning, OP)[10] 和写放大(write amplification, WA)[11] 等问题，ZNS SSD 可以有效提升 SSD 的读写吞吐，降低写入时的写放大，减少 SSD 本身的预留空间，并且还能解决传统 SSD 在空间占用达到一定程度时由于内部垃圾回收导致的性能不稳定的问题[12-13]，因此利用 ZNS SSD 来构建数据库系统是一个趋势[3].

图 1 展示了 ZNS SSD 和传统 SSD 数据放置对比，在 ZNS SSD 上数据由主机程序显式地控制放置. 虽然 ZNS SSD 具有如此多优点，但它同样带来了一些挑战[3,7]. 与传统基于闪存的 SSD 相比，ZNS SSD 移除了 FTL，将分区(Zone)的空间直接交由主机程序控制管理，由主机程序来处理垃圾回收、磨损均衡、数据放置等，这扩大了用户数据布局的设计空间. 由用户程序来决定数据的放置、生命周期和垃圾回收时机，通过有效合理地组织数据，可以提高系统的性能. 但 ZNS SSD 同样给主机程序的设计带来了新的要求，比如一个 Zone 内有一个写指针只能进行顺序写、不同 Zone 性能有差异、何时进行 Zone-Reset 操作等[7,14].

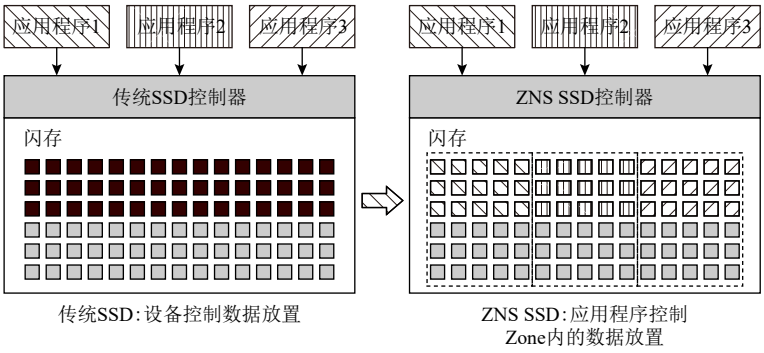

Fig. 1 Comparison of ZNS SSD and conventional SSD data placement

图 1 ZNS SSD 和传统 SSD 数据放置对比

B$^+$-tree 是数据库中经典的索引结构，以往研究者已经提出了多种针对 SSD、持久性内存等新型存储的 B$^+$-tree 优化方法[15-26]. 但是，已有的 SSD 索引设计重点在于减少对 SSD 的随机写操作. 虽然 ZNS SSD 的底层介质仍是闪存，但由于 Zone 内只能顺序写，因此随机写的问题不再是 ZNS SSD 上 B$^+$-tree 索引优化的第一目标. B$^+$-tree 如何在没有 FTL 的情况下适应 ZNS SSD 的分区特性以及 Zone 内顺序写要求，是 ZNS SSD 感知的 B$^+$-tree 索引需要解决的关键问题. 针对传统 SSD 设计的索引由于没有考虑 ZNS SSD 的特性，所以都无法直接运行在 ZNS SSD 上. 此外，根据我们的调研，目前也还没有提出 ZNS SSD 感知的索引结构.

本文提出了一种 ZNS SSD 感知的新型索引结构——ZB$^+$-tree(ZNS-aware B$^+$-tree). 我们发现，虽然 ZNS SSD 上要求 Zone 内顺序写，但是实际的分区块设备(zoned block device, ZBD)中除了只允许顺序写的顺序 Zone (sequential zone, Seq-Zone)，还存在着少量允许随机写的常规 Zone (conventional zone, Cov-Zone). 因此，我们结合了 ZNS SSD 内部这 2 类 Zone 的特性设计了 ZB$^+$-tree 的结构来适配 ZNS SSD. 具体而言，本文的主要工作和贡献可总结为 3 个方面：

1)提出了一种 ZNS SSD 感知的新型索引结构 ZB$^+$-tree. ZB$^+$-tree 利用了 ZNS SSD 内部的 Cov-Zone 来吸收对 Zone 的随机写操作，并将索引节点分散存储在 Cov-Zone 和 Seq-Zone 中，通过设计不同的节点结构来管理不同 Zone 上的节点，在保证 Zone 内顺序写要求的同时提高空间效率.

2)设计了 ZB$^+$-tree 上的相关操作，包括 Sync, Search, Insert, Delete 等，在保证索引性能的同时减少操作过程中的读写次数.

3)利用 null_blk[27] 和 libzbd[28] 模拟 ZNS SSD 设备开展实验，并将传统的 CoW B$^+$-tree 修改后作为对比索引. 结果表明，ZB$^+$-tree 能有效阻断级联更新，减少读写次数，在运行时间、空间利用率上均优于 CoW B$^+$-tree.

1 背景与相关工作

本节主要介绍目前业界对于 ZNS SSD 的相关研究和 SSD 感知的 B$^+$-tree 索引的相关工作，同时也介绍本文用于对比实验的 CoW B$^+$-tree.

1.1 ZNS SSD 相关工作

ZNS SSD 将其空间划分为许多的 Zones，在一个 Zone 内可以随机读，但是只能顺序写，当一个 Zone 写满时会触发 Zone-Reset 和 GC 操作. 图 2 显示了 Zone 的大致结构，每个 Zone 内有一个写指针，Zone 内有严格的顺序写限制.

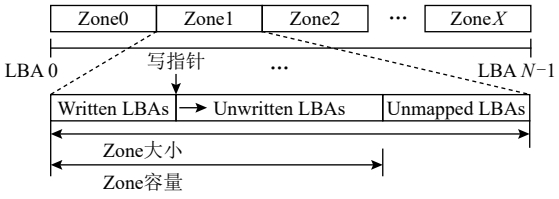

Fig. 2 The structure of Zone

图 2 Zone 的结构

传统 SSD 由于 FTL 的存在，会给主机程序提供块接口[1]，各类任务都交由 FTL 来处理，SSD 通过 FTL 提供给上层应用的接口是支持随机读写的，在程序设计时可以视为硬盘. FTL 的存在使得之前为硬盘设计的各类应用程序可以几乎不用修改而直接迁移到 SSD 上，但同时也导致了 SSD 需要预留空间、需要进行内部的垃圾回收、在空间占比达到一定程度时会发生性能抖动等问题[1,12].

ZNS SSD 相比于传统块接口的 SSD 有以下优点：首先在 ZNS SSD 上减少甚至移除了 FTL，将映射表的管理、垃圾回收和顺序写的限制都交给主机端进行控制，节约了成本，同时解决了预留空间的问题. 其次由于主机程序能控制 ZNS SSD 上数据的放置、垃圾回收时机，通过将不同特性的数据放置在不同的 Zone，选择合理的垃圾回收时机等策略将有助于提高系统性能、延长 SSD 使用寿命.

由于在 Zone 内只能顺序写，因此很多基于传统 SSD 抽象接口开发的应用和系统不能直接应用在 ZNS SSD 上[1-6]. 此外，由于 ZNS SSD 移除了设备内的垃圾回收，因此需要用户自行设计垃圾回收机制，带来了额外的复杂性. Stavrinos 等人[3] 分析了 ZNS SSD 的各项优势，并通过调研发现一旦行业因为 ZNS SSD 的成本和性能优势开始转向 ZNS SSD，则之前大多数 SSD 上的工作都需要重新审视，因此呼吁研究者转向 ZNS SSD 的研究. Shin 等人[7] 通过在 ZNS SSD 原型硬件上进行各种性能测试，为 ZNS SSD 上的软件设计提供了有效思路. Choi 等人[5] 提出了一种在 ZNS SSD 上的 LSM(log-structured merge)风格的垃圾回收机制，他们建议针对 ZNS SSD 进行细粒度的垃圾回收，并根据数据热度将细粒度数据存储在不同的分区内. 西部数据的 Bjørling 等人[1] 基于 ZNS SSD 开发出了 ZenFS 文件系统来适配 RocksDB 的文件接口，目前已经成功提交到 RocksDB 的社区. Han 等人[2] 在 ZNS 接口的基础上更进一步，提出了 ZNS+接口. ZNS+是一种对日志结构文件系统(log-structured file system, LFS)感知的 ZNS 接口，将用户定义的垃圾回收产生的有效数据拷贝放到设备内部进行，从而减少 I/O 操作，提升文件系统垃圾回收的效率.

虽然 ZNS SSD 具有如此多优点，但它同样带来了一些挑战. 与传统 SSD 相比，ZNS SSD 移除了 FTL，将 Zone 的空间直接交由主机程序控制管理，由主机程序来处理垃圾回收、磨损均衡、数据放置等，这扩大了用户数据存储管理的设计空间，也带来了设计上的困难. 总体而言，ZNS SSD 的顺序写和 Zone 划分等限制对现有的数据存储管理机制提出了诸多新的挑战，包括存储分配、垃圾回收、索引结构等.

1.2 SSD 感知的 B$^+$-tree 相关工作

在过去的 10 多年里，由于闪存技术的快速发展，学术界针对闪存感知的 B$^+$-tree 优化方法开展了广泛的研究. Roh 等人[15] 利用 SSD 内部的并行性来优化 B$^+$-tree 索引. Na 等人[16] 提出了一种动态页内日志风格的 B$^+$-tree 索引. Agrawal 等人[17] 设计了一种惰性更新的机制来使 B$^+$-tree 更适合 SSD 特性. Ahn 等人[18] 利用 CoW B$^+$-tree 的性质对 SSD 上的 B$^+$-tree 进行了

优化. Fang 等人[19]提出一种感知 SSD 持久度的 B$^+$-tree 方案. Jin 等人[20]则利用布隆过滤器(Bloom filter)和节点的更新缓存实现了不降低读性能的同时减少 SSD 写操作. Lv 等人[21]通过日志的方式将随机写转为顺序写来优化 SSD 上的 R-tree 的读写操作. Jin 等人[22]通过利用 SSD 内部的并行来批量处理小的写入, 提出了一种 flash 感知的线性散列索引. Ho 和 Park[23]通过在内存中设计一个写模式转换器, 将随机写转换为顺序写, 以此来充分利用 SSD 的特性. 文献 [24] 使用 IPL (in-page logging)和写缓存等技术设计符合 SSD 特性的 LSB$^+$-tree, 同时提高了索引的时间和空间效率.

总体而言, SSD 感知的 B$^+$-tree 索引的设计重点在于减少对 SSD 的随机写操作. 这是因为 B$^+$-tree 本身是一种写不友好的索引, 因此对于 SSD 上的写密集应用性能较差. ZNS SSD 的介质仍是闪存, 因此同样需要考虑减少随机写操作. 但由于 Zone 内只能顺序写, 随机写的问题不再是 ZNS SSD 上 B$^+$-tree 索引优化的第一目标. 如何适应数据的分区存储和顺序写特性, 是设计 ZNS SSD 感知的 B$^+$-tree 索引需要重点考虑的问题. 就目前的研究进展, 还没有一种已有的 SSD 索引能够直接运行在 ZNS SSD 上.

1.3 CoW B$^+$-tree

目前学术界还没有提出针对 ZNS SSD 的索引结构. 与本文工作较相关的已有工作主要是 CoW B$^+$-tree. CoW B$^+$-tree 是一种追加写(append-only write)的 B+树结构[25]. 这刚好适应了 ZNS SSD 的 Zone 内顺序写要求, 因此如果不考虑 Zone 内数据存储分配的话, 可以将 CoW B$^+$-tree 修改后运行在 ZNS SSD 上. 图 3 给出了 CoW B$^+$-tree 的数据更新过程.

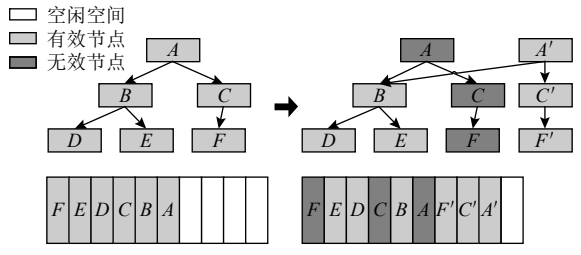

Fig. 3 An example of CoW B$^+$-tree update

图 3 CoW B$^+$-tree 更新过程示例

文献 [18] 对 Cow B$^+$-tree 针对 SSD 特性做了一些优化以减少写放大, 但是却无法保证 B$^+$-tree 节点大小相同, 因此本文还是选择传统 CoW B$^+$-tree[25]进行对比, CoW B$^+$-tree 算法流程和传统的 B$^+$-tree 基本一样, 只是修改叶节点时由于不能原地更新, 因此需要把修改后的节点直接附加在写指针处, 由于叶节点的地址发生了改变, 因此还需要修改其父节点中指向叶节点的指针. 这一修改会级联往上传递, 直到根节点.

但是, 直接将 CoW B$^+$-tree 的所有节点都写在一个 Zone 中将会很快耗尽 Zone 的空间, 从而频繁触发 Zone-Reset 操作. 因此, 在本文后续的对比实验中, 我们对 CoW B$^+$-tree 进行了修改使其能够利用所有的 Zone, 并减少频繁的 Zone-Reset 操作. 由于主机程序可以得知 Zone 的使用情况, 我们总是将 CoW B$^+$-tree 节点写入剩余空间最多的 Zone, 并根据当前 Zone 的使用情况动态选择 Zone 而不是固定写入一个 Zone. 这样可以在使用该索引时尽量减少 Zone-Reset, 同时充分利用 ZNS SSD 的空间.

2 ZB$^+$-tree 索引结构

2.1 设计思路

由于 ZNS SSD 具有多分区特性, 不同的 Zone 性能有所差异, 在频繁访问的磨损较多的 Zone 会有更高的访问延迟[7], 且在 Seq-Zone 中有严格的顺序写要求. 当 Zone 写满时会触发 Zone-Reset 操作和垃圾回收, 这 2 个操作非常耗时, 会带来性能的抖动. 因此 ZB$^+$-tree 主要针对 4 个目标进行设计: 1)索引要能充分利用 ZNS SSD 的多分区特性; 2)要满足在 Seq-Zone 中的严格顺序写限制; 3)将频繁访问的数据尽量放置在磨损较少的 Zone 以降低访问延迟; 4)尽量减少在运行过程中的 GC 和 Zone-Reset 操作, 以避免性能抖动.

针对上述目标, ZB$^+$-tree 进行了全新设计, 主要设计思路总结为 5 个方面:

1)将索引节点分散在多个 Zone 中, 允许节点在不同 Zone 之间进行移动. 由于 Cov-Zone 中允许进行原地更新, 如果能充分利用这一特性, 就能将对索引的原地更新吸收在 Cov-Zone 中. 当节点写满时则整体移动到 Seq-Zone 中, 既保证不会有太大的写放大, 而同时也能保证在 Seq-Zone 中的顺序写要求.

2)将对 Seq-Zone 中节点的更新也吸收在 Cov-Zone 中, 为 Seq-Zone 中不可原地更新的节点设计了日志节点结构, 日志节点放置在 Cov-Zone 中以进行原地更新, 减少写放大. 同时为避免日志节点过多而影响整体的读写性能, 需要将日志与节点合并, 可通过设计新的 Sync 操作来实现.

3)为不同类型的节点动态选择不同的 Zone 进

行放置，由于叶节点和内部节点更新频率不同，叶节点频繁更新而内部节点相对更新较少，因此我们提出一种动态选择 Zone 的策略，将频繁更新的叶节点放置在空间占用较少、磨损较少的 Zone 中，而将内部节点放置于空间占用和磨损相对较多的 Zone 中，以充分利用 ZNS SSD 的多分区特性，并降低访问延迟.

4）为管理分散在不同 Zone 的节点和对应的日志节点，我们为叶节点和内部节点都设计相应的 head 节点，以记录节点的状态、地址、日志情况，并将 head 节点存储于 Cov-Zone 中，以阻断级联更新.

5）由于不同分层的节点处于不同类型的 Zone 中，对节点的合并控制也将采用不同的策略，对于都在 Cov-Zone 中的节点，采用严格控制合并策略；对于节点分散在 Cov-Zone 和 Seq-Zone 中的节点，则采用机会主义合并策略. 通过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和写放大.

2.2 ZB+-tree 总体结构

在本文的设计中，ZB+-tree 总共分为 4 层，从上到下分别是 IH 层、Interior 层、LH 层、Leaf 层，其中 IH 层由 interior-head 节点组成，LH 层由 leaf-head 节点组成，Interior 层由 interior 节点和 interior-log 节点组成，Leaf 层由 leaf 节点和 leaf-log 节点组成. 由于通常来说 B+-tree 为 3~4 层，我们设计的 IH 可以视为 B+-tree 的根节点. IH 层和 LH 层的节点既可以作为 B+-tree 的内部节点（里面存着 key 值和子节点地址），同时又记录着下一层的节点状态和日志情况. 图 4 展示了 ZB+-tree 的逻辑结构，其中 F 代表 key 值为 First（key 无法取到的最小值），K 代表普通 key 值.

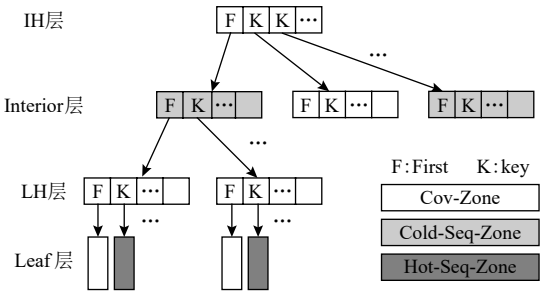

Fig. 4 ZB+-tree logical structure

图 4 ZB+-tree 逻辑结构

由于主机程序可以随时知道 ZNS SSD 上各个 Zone 的使用情况，这里提出一种动态选择 Zone 的方法，即当有节点需要从 Cov-Zone 移入到 Seq-Zone 中时，如果是 leaf 节点，则动态选择当前剩余容量最多的 Zone（称为 Hot-Seq-Zone）放置，如果是 interior 节点，则动态选择当前剩余容量最少的 Zone（称为 Cold-Seq-Zone）放置. 之所以这么区分，是因为叶节点和内部节点的更新频率不同，把更新频率高的叶节点往剩余空间最大的 Zone 内放置可以避免由于 Zone 空间被迅速占满而导致的 Zone-Reset 操作. 同时剩余容量多、磨损较少的 Zone 的读写性能也更好，可以便于 leaf 的频繁读写.

图 5 显示了 ZB+-tree 节点在 ZNS SSD 上的具体分布情况，其中 IH 和 LH 层都位于 Cov-Zone 中，而 interior 节点和 leaf 节点则分布在 Cov-Zone 和 Seq-Zone 中. 在本文中，我们约定：白色代表 Cov-Zone，深灰色代表 Hot-Seq-Zone，浅灰色代表 Cold-Seq-Zone，灰色渐变代表处于 Seq-Zone 中可能是 Hot 状态也可能是 Cold 状态.

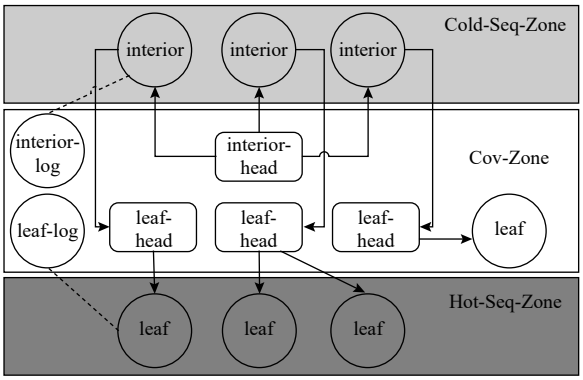

Fig. 5 Distribution of ZB+-tree nodes on ZNS SSD

图 5 ZB+-tree 节点在 ZNS SSD 上的分布

对于 interior 节点和 leaf 节点，设计了相应的状态转换机制，如图 6 所示. ZB+-tree 的 interior 节点和 leaf 节点有 4 种可能的状态.

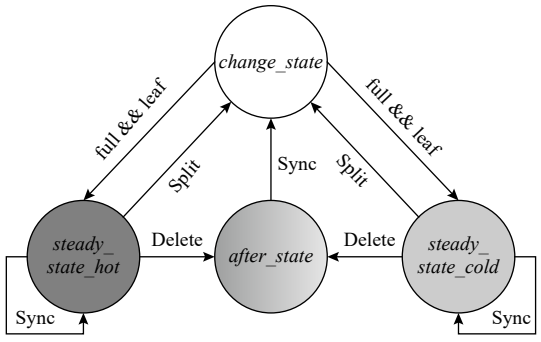

Fig. 6 ZB+-tree node state transition

图 6 ZB+-tree 节点状态转换

1）*change_state*. 处于 *change_state* 的节点都位于 Cov-Zone 中，并且节点未满，可以就地插入、删除、更新. 处于 *change_state* 的节点一旦满了，如果是叶节点，则整体移动到 Hot-Seq-Zone 中；如果是内部节点，

则整体移动到 Cold-Seq-Zone 中，这刚好符合 ZNS SSD 的顺序写限制.

2) *steady_state_hot*. 节点位于 Hot-Seq-Zone 中，且为叶节点，节点一定是满的，并且不可就地更改. 对此状态的节点进行插入将导致其分裂成 2 个 *change_state* 的叶节点，对此状态节点的更新和删除将记录在 leaf-log 节点中.

3) *steady_state_cold*. 节点位于 Cold-Seq-Zone 中，且是内部节点，节点一定是满的，并且不可就地更改. 对此状态的节点进行插入将导致其分裂成 2 个 *change_state* 的内部节点，对此状态节点的更新和删除将记录在 interior-log 节点中.

4) *after_state*. 处于该状态的节点一定带有 log 节点，并且 log 节点中有删除记录，表示该节点在 *steady_state* 经过了删除. 对此状态的节点进行插入会首先触发 Sync 操作，将节点和对应的 log 节点合并之后再进行插入，对此状态节点的更新和删除将记录在 log 节点中.

处于 *steady_state* 的节点可能带 log 节点也可能不带 log 节点，log 节点都位于 Cov-Zone 中以吸收对 Seq-Zone 中节点的原地更新，对 *steady_state* 的节点进行的更新和删除操作会直接写到对应的 log 节点里(如果该节点原本不带 log 节点则为其分配一个 log 节点). 处于 *steady_state* 的节点可以进行 3 种操作：

1) Sync 操作. 将节点与对应的 log 节点进行合并，Sync 操作完成之后节点还是处于 *steady_state*.

2) Delete 操作. 在 log 节点中增加删除记录，Delete 操作完成之后节点转换为 *after_state* 表示节点已经经过了删除.

3) Split 操作. 节点分裂成 2 个，Split 操作完成之后节点转化为 2 个 *change_state* 的节点并移动到 Cov-Zone 中.

处于 *after_state* 的节点可以进行 Sync 操作，将节点与对应的 log 节点进行合并. 由于必然经过了删除操作，实际上 *after_state* 的节点在逻辑上代表一个未满的节点，因此合并后节点状态转换为 *change_state*.

2.3 ZB$^+$-tree 节点结构设计

leaf 节点设置为 n 个 key 值和 n 个 value 值，leaf-log 节点的结构与 leaf 节点的结构完全相同，之所以设置为结构相同，是因为在 Sync 操作中需要将 leaf-log 节点直接转化为 leaf 节点(具体细节见 3.1 节 Sync 操作)，在 leaf-log 节点中通过位置与 leaf 节点中的键值对应，如图 7 所示.

对于 leaf-head 节点，结构为 n+1 个 key 值和 n+1 个 ptr 结构，如图 8 所示. ptr 结构中包括 3 个值：

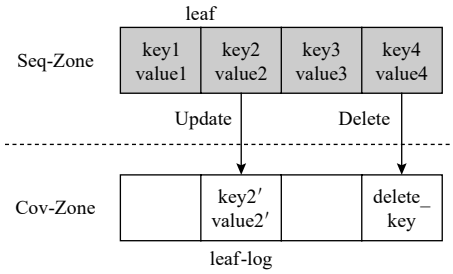

Fig. 7 An example of leaf and corresponding leaf-log

图 7 leaf 和对应的 leaf-log 示例

1) *state*. 表示子节点所处的状态.

2) *addr*. 表示子节点所在的位置.

3) *log_addr*. 表示子节点对应的日志节点所处的位置.

leaf-head 节点的第 1 个 key 值一定是 First(key 无法取到的最小值).

interior 节点的结构为 n+1 个 key 和 n+1 个 addr(子节点的地址)，interior-log 和 interior 结构完全相同(同样是因为 Sync 操作的要求)，interior-head 的结构和 leaf-head 的结构相同. 在 interior 和 interior-head 节点中，第 1 个 key 值同样设置为 First. 之所以在 interior 节点中引入 First 值是因为在 ZB$^+$-tree 中的 Sync 操作需要让日志节点和 interior 节点保持相同结构以便于 log 节点与正常的内部节点之间直接进行转化，而日志节点又是通过位置与原内部节点进行对应，所以需要给第 1 个位置一个 key 值，代表这是第 1 个子节点. 之所以在 leaf-head 和 interior-head 中引入 First 值，是因为对于每一个 leaf 节点或者是 interior 节点，都需要相应的 ptr 结构来指示其状态、位置和日志情况. 即使树中只有一个 leaf 节点或只有一个 interior 节点也要有对应的 ptr 结构，因此，需要为第 1 个 ptr 结构设置 key 值为 First，表示这是子树中第 1 个 interior 节点或者第 1 个 leaf 节点，而其他的 key 值则与正常 B$^+$-tree 中的 key 值含义相同，代表对应子树中的最小 key 值.

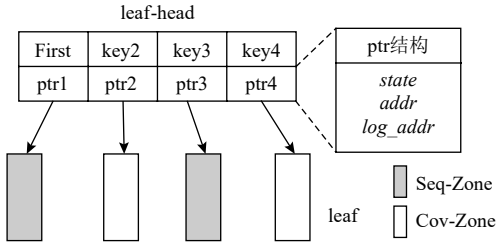

Fig. 8 The structure of leaf-head

图 8 leaf-head 的结构

3 ZB+-tree 索引操作

本节主要介绍 ZB+-tree 的 Sync, Search, Insert, Delete 等操作，并给出了时间性能和空间代价分析.

3.1 Sync

在 ZB+-tree 中无论是 interior-log 节点还是 leaf-log 节点都只记录更新和删除操作，其结构和对应的 interior 节点或 leaf 节点相同，当树中存在的日志节点过多时会对 Cov-Zone 有较大的占用而且会影响整体的查找性能，因为读一个节点之后还需要再读一次日志节点来重建最新的节点. 因此需要将日志节点和对应的节点进行合并，构建最新的节点并释放日志节点空间，我们称之为 Sync 操作. 在日志节点写满时同样需要将日志与节点进行合并. 在 ZB+-tree 中，我们设计了 2 种方式的合并，当日志节点未满时的合并称之为 Normal_apply，当日志节点满了时的合并称之为 Switch_apply.

1）Normal_apply. 图 9 给出了 Normal_apply 的过程. 当节点对应的 log 节点还未满时，如果节点处于 *steady_state*，表示节点还未进行过删除操作，log 节点中无删除记录，因此根据 log 节点进行更新后的节点仍然处于 *steady_state*，此时将新的节点写回 Seq-Zone 并释放 Cov-Zone 上 log 节点的空间. 如果节点处于 *after_state*，则表示节点经过了删除操作，log 节点中一定有删除记录，根据 log 节点进行更新后的节点转换成 *change_state* 并写到 Cov-Zone 中原 log 节点的位置. 算法 1 给出了 Normal_apply 操作流程.

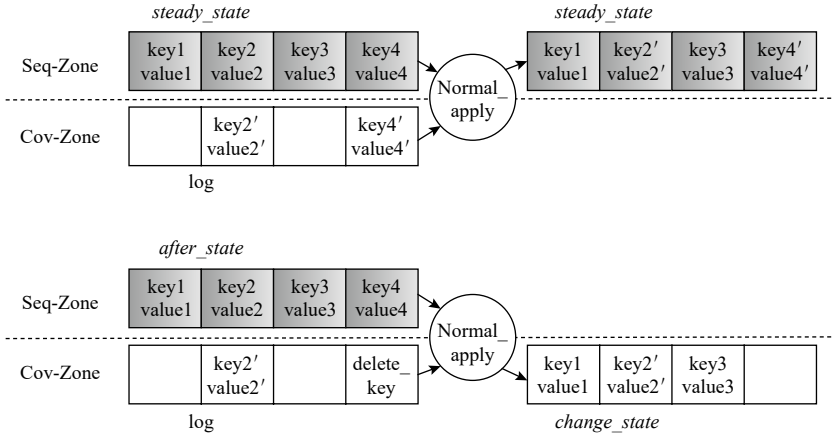

Fig. 9 Normal_apply schematic

图 9 Normal_apply 示意图

算法 1. *Normal_apply_leaf*(&*leaf*, &*leaf_log*, &*leaf_ptr*).

输入：叶节点 *leaf*, 对应的日志节点 *leaf_log*, 叶节点对应的 ptr 结构 *leaf_ptr*;

输出：无.

① if *leaf_ptr*. *state* == *after_state then*
② for each 〈*key*, *value*〉 in *leaf_log*
③ if *key* ≠ *delete_key* then
④ update 〈*key*, *value*〉 in *leaf*;
⑤ else delete 〈*key*, *value*〉 in *leaf*;
⑥ end if
⑦ end for
⑧ *write_leaf*(*leaf_ptr*. *log_addr*, *leaf*);
⑨ *leaf_ptr*. *state* = *change_state*;
⑩ *leaf_ptr*. *addr* = *leaf_ptr*. *log_addr*;
⑪ *leaf_ptr*. *log_addr* = Null;
⑫ end if
⑬ if *leaf_ptr*. *state* == *steady_state_hot* then
⑭ for each 〈*key*, *value*〉 in *leaf_log*
⑮ update 〈*key*, *value*〉 in *leaf*;
⑯ end for
⑰ reclaim *leaf-log* ;
⑱ *addr* = select the Zone with minimum capacity;
⑲ *write_leaf*(*addr*, *leaf*);
⑳ *leaf_ptr*. *addr* = *addr*;
㉑ *leaf_ptr*. *log_addr* = Null;
㉒ end if

2）Switch_apply. 图 10 给出了 Switch_apply 的过程. 当节点对应的 log 节点已经写满时，将发生 Switch_apply，与 Normal_apply 不同的是，Switch_apply 不需要读原节点，而只需要读其 log 节点，因为日志已满说明原节点中所有键值对都已经经过了更新或者删

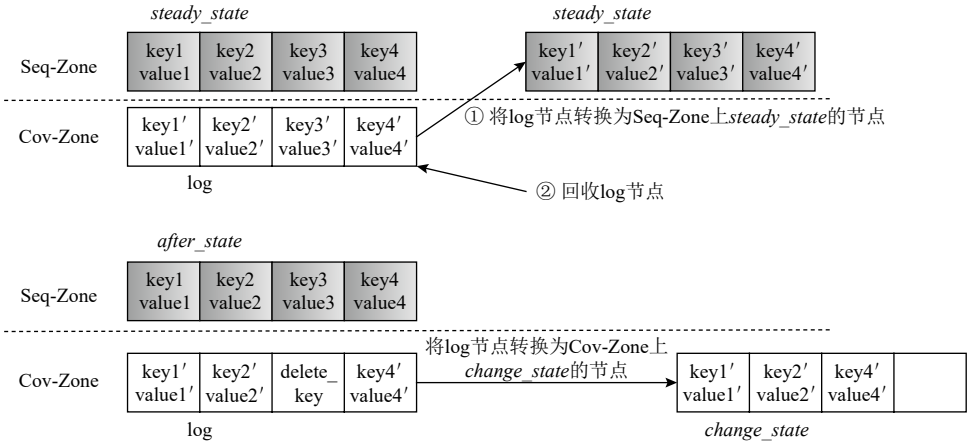

Fig. 10 Switch_apply schematic

图 10 Switch_apply 示意图

除. 如果原节点处于 *steady_state*, 说明对节点的所有键值对都进行过更新操作, 因此直接将 log 节点作为新的节点写入 Seq-Zone 中, 并释放在 Cov-Zone 中 log 节点的空间. 如果原节点处于 *after_state*, 说明其除了进行过更新操作之外还有删除操作, 因此把 log 节点中有删除标记(delete_key)的键值对去除后, 转化为 *change_state* 的节点, 直接写回 Cov-Zone 中原 log 节点位置. 算法 2 给出了 Switch_apply.

算法 2. *Switch_apply_leaf*(&*leaf_log*, &*leaf_ptr*).

输入：叶节点对应的日志节点 *leaf_log*, 叶节点对应的 ptr 结构 *leaf_ptr*;

输出：重建的 *leaf*.

① if *leaf_ptr. state* == *steady_state_hot* then
② *addr* = select the Zone with minimum capacity;
③ *write_leaf*(*addr*, *leaf_log*);
④ reclaim *leaf-log*;
⑤ *leaf_ptr. addr* = *addr*;
⑥ *leaf_ptr. log_addr* = Null;
⑦ return *leaf-log*;
⑧ end if
⑨ if *leaf_ptr. state* == *after_state* then
⑩ for each 〈*key*, *value*〉 in *leaf_log*
⑪ if *key* == *delete_key* then
⑫ delete 〈*key*, *value*〉 in *leaf-log*;
⑬ end if
⑭ end for
⑮ *write_leaf*(*leaf_ptr. log_addr*, *leaf_log*);
⑯ *leaf_ptr. state* = *change_state*;
⑰ *leaf_ptr. addr* = *leaf_ptr. log_addr*;
⑱ *leaf_ptr. log_addr* = Null;
⑲ return *leaf-log*;
⑳ end if

Sync 操作(见算法 3)对于 interior 节点和 leaf 节点都类似, 唯一区别在于 interior 节点中为 *n*+1 个键值对而 leaf 节点中为 *n* 个键值对. 对 leaf 节点的操作为 Sync, 对于 interior 节点也有相应的 in_Sync 操作.

算法 3. *Sync*(&*leaf*, &*leaf_log*, &*leaf_ptr*).

输入：叶节点 *leaf*, 对应的日志节点 *leaf_log*, 叶节点对应的 ptr 结构 *leaf_ptr*;

输出：重建的 *leaf*.

① if *leaf_log* is full then
② return *Switch_apply_leaf*(*leaf_log*, *leaf_ptr*);
③ else
④ *Normal_apply_leaf*(*leaf*, *leaf_log*, *leaf_ptr*);
⑤ return *leaf*;
⑥ end if

3.2 Search

在 ZB$^+$-tree 上的 Search 过程如算法 4 所示, 如果 ZB$^+$-tree 中还没有 IH 层, 也就是整棵树只有 2 层(LH 层和 Leaf 层)时, 只有一个 leaf-head 节点, 此时直接从 leaf-head 开始进行搜索. 如果树中存在 IH 层, 则根据要搜索的 key 值, 先从 IH 中得到 interior 节点对应的 ptr 结构, 如果 interior 节点没有 log 节点, 则直接读出 interior 节点; 如果 interior 节点有 log 节点, 则会触发 *apply_innner_log*()操作, 这个操作只是在内存中重建最新的 interior 节点, 并不会对 ZNS SSD 上的 interior 节点和日志节点进行修改. 这样做的目的是避免在 Search 过程中触发写操作. 得到了最新的 interior 节点之后, 在其中根据 key 值搜索得到 leaf-head 节点的地址, 并读出 leaf-head 节点, 再从 leaf-head 开始搜索,

即触发 *Search_leaf_head*()操作，具体见算法 5. 其过程与从 IH 搜索 interior 节点的过程类似，其中 *apply_leaf_log*()也只是在内存中建立最新的 leaf 节点，并不修改 ZNS SSD 上的 leaf 节点和 log 节点. 得到最新的 leaf 节点之后，再根据 key 值进行搜索，如果搜到则返回 value 值，否则返回 Null，表示没有搜到.

算法 4. *Search*(*key*).

输入：要搜索的 *key* 值；

输出：搜到返回对应的 *value* 值，否则返回 Null.

① if IH is empty then
② return *Search_leaf_head*(*leaf_head_last*, *key*)；
③ else
④ *inter_ptr* = search IH to get the *interior_ptr*；
⑤ if *inter_ptr. log_addr* == Null then
⑥ *inter* = *read_interior*(*inter_ptr. addr*)；
⑦ else
⑧ get the *log* and *old_inter*；
⑨ *inter* = *apply_inner_log*(*old_inter*,*log*)；
⑩ end if
⑪ *leaf_head_addr* = search the *inter* to get the address of leaf_head node；
⑫ *leaf_head* = *read_LH*(*leaf_head_addr*)；
⑬ return *Search_leaf_head*(*leaf_head*, *key*)；
⑭ end if

算法 5. *Search_leaf_head* (*leaf_head*, *key*).

输入：要搜索的 *leaf_head* 节点，要搜索的 *key* 值；

输出：搜到返回对应的 *value* 值，否则返回 Null.

① *leaf_ptr* = get the ptr struct of leaf node from *leaf_head*；
② if *leaf_ptr. log_addr* == Null then
③ *leaf* = *read_leaf*(*leaf_ptr. addr*)；
④ else
⑤ *log* = *read_leaf_log*(*leaf_ptr. log_addr*)；
⑥ *old_leaf* = *read_leaf* (*leaf_ptr. addr*)；
⑦ *leaf* = *apply_leaf_log*(*old_leaf*, *log*)；
⑧ end if
⑨ for each 〈*KEY, VALUE*〉 in *leaf* do
⑩ if *key* == *KEY* then
⑪ return *VALUE*；
⑫ end if
⑬ end for
⑭ return Null.

3.3 Insert

在 ZB+-tree 上的 Insert 操作会先根据要插入的 key 值从根节点一路搜索到叶节点对应的 leaf-head 节点，搜索过程与 Search 相同，从 leaf-head 节点中得到对应的 ptr 结构，如果叶节点处于 *change_state*，则直接往叶节点中插入键值对，如果满了则移动到 Seq-Zone 中. 而对于 *steady_state* 的叶节点，如果不带 log 节点，则直接进行 Split 操作，分裂成 2 个 Cov-Zone 上的 *change_state* 的节点，如果带 log 节点，则先进行 Sync 操作，合并 leaf 节点和 log 节点之后再插入键值对，Insert 具体细节见算法 6，ZB+-tree 上的 Split 操作和正常 B+-tree 的操作类似，唯一需要注意的是对树中 First 值的维护.

算法 6. *Insert*(*key*, *value*).

输入：*key*, *value* 是要插入的键值对；

输出：无.

① *leaf_head* = find the leaf_head node of the corresponding leaf node；
② *leaf_ptr* = get the ptr struct of the leaf node；
③ *leaf* = get the leaf node；
④ if *leaf_ptr. state* == *change_state* then
⑤ insert 〈*key*, *value*〉 in *leaf*；
⑥ if *leaf* is full then
⑦ move *leaf* to Seq-Zone；
⑧ *leaf_ptr. state* = *steady_state_hot*；
⑨ *leaf_ptr. addr* = *new_addr*；
⑩ end if
⑪ else
⑫ if *leaf_ptr. log_addr* == Null then
⑬ *Split*(*leaf*, *key*, *value*)；
⑭ else
⑮ *log* = get the leaf_log node of the *leaf*；
⑯ *leaf* = *Sync*(*leaf*, *log*, *leaf_ptr*)；
⑰ if *leaf_ptr. state* == *change_state* then
⑱ insert 〈*key*, *value*〉 to *leaf*；
⑲ if *leaf* is full then
⑳ move *leaf* to Seq-Zone；
㉑ *leaf_ptr. state* = *steady_state_hot*；
㉒ *leaf_ptr. addr* = *new_addr*；
㉓ end if
㉔ end if
㉕ if *leaf_ptr. state* == *steady_state_hot* then
㉖ *Split*(*leaf*, *key*, *value*)；
㉗ end if
㉘ end if
㉙ end if

3.4 Delete

ZB$^+$-tree上的删除操作与经典B$^+$-tree算法有许多不同之处，由于不同层的节点所处的Zone性质是不同的，对于不同层的节点在进行Delete操作时，也将采用不同的控制合并的策略.

对于IH层，由于所有的leaf-head节点都处于Cov-Zone中，可以发生原地更新，因此在删除时当节点容量达到下限(lower_bound)，我们采取严格控制合并的策略，即leaf-head节点的容量严格控制在lower_bound和full之间(整棵树只有一个leaf-head节点时例外). 而对于Leaf层，在进行删除时我们采取的是机会主义策略控制合并，即查看邻居节点，能合并就合并(二者容量之和小于n)，不能合并就不作处理，能满足合并条件则意味着二者必然都处于*change_state*，即都在Cov-Zone中. 之所以采取不同策略，是因为Leaf层的节点分散在Seq-Zone和Cov-Zone中，且节点可能带log节点也可能不带log节点. 如果不能发生合并，从*steady_state*的邻居借一个键值对，并不会减少空间的占用，反而将引发级联的修改，还要处理log节点. 在机会主义策略控制下，leaf节点的容量范围为1~n.

对于Interior层的节点我们同样采取机会主义控制合并的策略，理由和Leaf层相同，interior节点的容量范围为1~n+1，当节点只有一个key值时必然是First，算法7展示了对Leaf层的Delete操作.

算法7. *Delete*(*key*).

输入：*key*是要删除的键值对的key值；

输出：删除成功返回true，删除失败返回false.

① *leaf_head* = find the leaf_head node of the corresponding leaf node;
② *leaf_ptr* = get the ptr struct of the leaf node;
③ *leaf* = get the leaf node;
④ if *find_in_leaf*(*key*, *leaf_ptr*) == false then
⑤ return false;
⑥ else
⑦ if *leaf_ptr. state* == *change_state* then
⑧ *delete_in_leaf*(*leaf*, *key*);
⑨ if *leaf* is empty then
⑩ *LH_delete*(*LH*, *key*);
⑪ reclaim *leaf* in Cov-Zone;
⑫ return true;
⑬ else do opportunism merge;
⑭ end if
⑮ end if
⑯ if *leaf_ptr. state* == *after_state* then
⑰ *log* = get the log of *leaf*;
⑱ Insert *delete_key* to *log*;
⑲ return true;
⑳ else
㉑ if *leaf_ptr. log_addr* == Null then
㉒ Allocate a new *log* for *leaf*;
㉓ Insert *delete_key* to *log*;
㉔ *leaf_ptr. state* = *after_state*;
㉕ *leaf_ptr. log_addr* = *log_addr*;
㉖ return true;
㉗ else
㉘ *log* = get the log of *leaf*;
㉙ Insert *delete_key* to *log*;
㉚ *leaf_ptr. state* = *after_state*;
㉛ return true;
㉜ end if
㉝ end if
㉞ end if

3.5 理论代价分析

3.5.1 时间性能

4层的ZB$^+$-tree在查询时最坏情况下需要读6次，即读interior-head节点、读interior节点和interior-log节点、读leaf-head节点、再读leaf节点和leaf-log节点，如果Interior层和Leaf层没有log节点，则与4层的CoW B$^+$-tree一样，需要读4次. 在进行插入操作时，CoW B$^+$-tree需要先4次读出根到叶节点的路径，如果不发生分裂需要4次写操作，在最坏情况下除了根节点外每层节点都分裂，则需要7次写操作. 而ZB$^+$-tree插入时同样需要先进行4~6次读，如果不发生分裂则只需要1次写操作(即直接往Cov-Zone中就地插入)，如果发生分裂，最坏情况下需要7次写操作，但大多数情况下不会出现每一层都恰好分裂，因此往往只需要1~2次写操作即可，即将级联的更新阻断在Cov-Zone中. 对CoW B$^+$-tree的删除操作，同样需要先进行4次读，随后如果没发生合并，最好情况下需要4次写操作，如果发生合并或者向邻居借键值对则需要更多的读写. 而ZB$^+$-tree在进行4~6次读操作之后，如果没发生合并，则只需要1次写操作(*change_state*的节点就地删除后只需要1次写；*steady_state*的节点则往log节点中插入删除记录，也只需要1次写)，如果发生合并，则在机会主义策略控制合并的2层进行的合并操作会更少，并且在这2层不会发生向邻居借键值对的操作，而在LH层和

IH 层与 CoW B+-tree 一样都采用了严格控制合并的策略，因此总体上的读写次数也会少于 CoW B+-tree. 结合上述分析，可知 ZB+-tree 在查询性能上与 CoW B+-tree 接近，在插入时会减少写操作，而在删除时会同时减少读写操作.

3.5.2 空间代价

CoW B+-tree 的每次更新都至少需要将从根到叶节点的路径重新写回到 ZNS SSD 中；而 ZB+-tree 则将更新吸收在 Cov-Zone 中，往往只需要在 Cov-Zone 中进行一次原地更新. 因此，相比于 CoW B+-tree, ZB+-tree 会大大减少对 Seq-Zone 的占用.

假设一个节点大小为 m, 一个 4 层的 ZB+-tree, 阶数为 n, 最坏情况下，每个 leaf 节点都带有 leaf-log 节点，且每个 interior 节点都带有 interior-log 节点，此时有效节点总大小可估计为

$$[2(n+1)^3+(n+1)^2+2(n+1)+1]\times m. \quad (1)$$

1 棵 4 层的 CoW B+-tree 有效节点所占空间大小即正常 B+-tree 所占空间大小，总空间大小可估计为

$$[(n+1)^3+(n+1)^2+(n+1)+1]\times m. \quad (2)$$

随着对索引的修改，CoW B+-tree 会不断将修改过的节点以及到根节点的路径上的节点写入 SSD 中. 上述分析仅仅针对其有效节点部分，实际运行中 CoW B+-tree 还会产生大量无效节点，进一步增加空间占用率. 而 ZB+-tree 将原地更新吸收在 Cov-Zone 中，实际运行时虽然也会在 Seq-Zone 中产生无效节点，但数量会远远少于 CoW B+-tree, 且其有效节点部分在最坏情况下也和 CoW B+-tree 处于同一个量级. 因此，总体上，ZB+-tree 的空间代价低于 CoW B+-tree.

ZB+-tree 相比于 CoW B+-tree 在 LH 层和 IH 层增加了一些数据结构用于管理下一层的节点，由于 IH 层和 LH 层都是内部节点，相对较少，因此额外的空间代价并不会特别高. ZB+-tree 对 Cov-Zone 的占用情况则主要和工作负载有关，需要通过实验来进行验证.

4 实验与分析

4.1 实验设置

服务器操作系统为 Ubuntu20. 04. 1LTS, 内核版本 5. 4. 0, gcc 版本为 9. 4. 0 . 由于目前还没有商用 ZNS SSD, 因此我们使用了 null_blk 来模拟 ZNS SSD 设备，并使用西部数据的 ZBD 操作库 libzbd 进行实验. 具体的实验配置如表 1 所示.

由于目前还没有提出 ZNS SSD 感知的索引，因此我们将 CoW B+-tree 进行了修改使其能够运行在 ZNS SSD 上. 具体而言，总是将 CoW B+-tree 节点写入剩余空间最多的 Zone, 并根据当前 Zone 的使用情况动态选择要写入的 Zone, 以尽量减少 Zone-Reset 操作，同时充分利用 ZNS SSD 的空间.

Table 1 Experimental Configuration

表 1 实验配置

服务器组件	说明
OS	Ubuntu 20.04.1 LTS
CPU	AMD EPYC 7 742 64-Core@2.60GHz
DRAM	256GB DDR4（2 666 MHz）
GCC version	9.4.0
libzbd version	2.0.3

实验使用 YCSB[29] 作为测试负载. YCSB 是目前在存储和数据库领域广泛使用的测试负载，它也允许用户自行配置读写比例和访问倾斜性. 在实验中我们利用 YCSB 生成了 5 个测试负载，说明如下：

1）Workload1 是写密集的负载，包含插入、删除、查找操作，并且 3 类操作的占比分别为插入 40%、删除 30%、查找 30%.

2）Workload2 是读密集的负载，包含插入、删除、查找操作，并且 3 类操作的占比分别为插入 10%、删除 10%、查找 80%.

3）Workload3 是读写均衡的负载，包含插入、删除、查找操作，并且 3 类操作的占比分别为插入 25%、删除 25%、查找 50%.

4）Workload4 是纯写负载，包含插入、删除操作，并且 2 类操作的占比分别为插入 50%、删除 50%.

5）Workload5 是纯读负载，只包含查找操作.

我们在 5 种负载上分别测试不同数据量和不同数据分布下的性能. 数据量分别设置为 50 万、150 万、250 万键值记录对，数据分布采用了文献 [29] 中的 3 类分布，包括 Zipfian 分布、uniform 分布、latest 分布. 在 YCSB 中执行测试前会先加载数据，以下的实验数据都是指在数据加载完成之后再进行各种操作时统计的数据. CoW B+-tree 实验中模拟的 ZBD 设备为 40 个 Seq-Zone 以及 1 个 Cov-Zone(为保证实验对比公平，我们把 Cov-Zone 按照 Seq-Zone 的顺序写模式来使用)，每个 Zone 大小为 2 GB; ZB+-tree 实验中模拟的 ZBD 设备为 40 个 Seq-Zone 以及 1 个 Cov-Zone, 大小都为 2GB. 在模拟器中块大小统一设置为 4 KB.

4.2 实验结果分析

4.2.1 运行时间比较

图 11~13 给出了 ZB+-tree 和 CoW B+-tree 的运行时间对比. 可以看到，在不同数据规模和不同工作负

载下, ZB$^+$-tree 运行时间都要小于 CoW B$^+$-tree. 虽然 ZB$^+$-tree 相对于 CoW B$^+$-tree 在查询时可能要多读 1~2 次 log 节点, 但由于在插入和删除操作中减少了读写次数, 且 SSD 读操作比写操作要快[7], 因此 ZB$^+$-tree 总体性能要好于 CoW B$^+$-tree, 与理论分析相符.

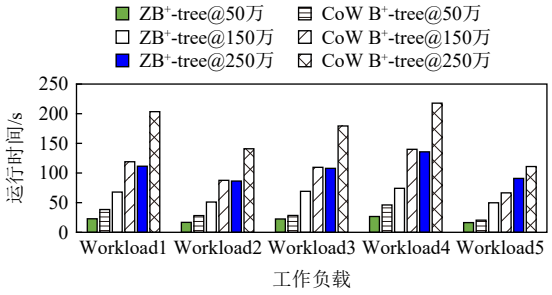

Fig. 11 Running time comparison under the Zipfian distribution

图 11 Zipfian 分布下运行时间对比

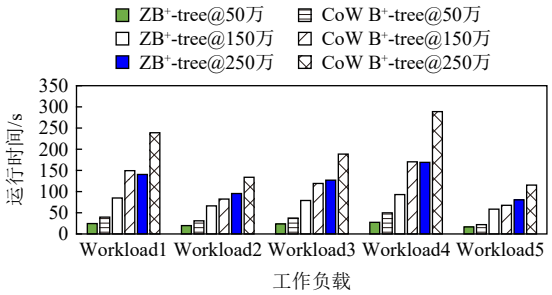

Fig. 12 Running time comparison under the uniform distribution

图 12 uniform 分布下运行时间对比

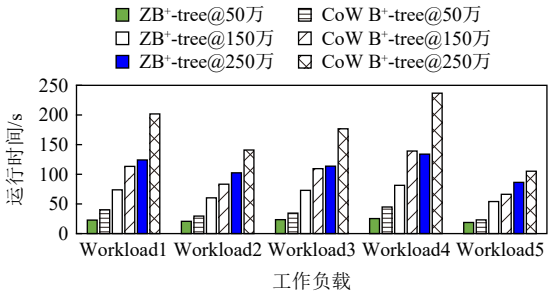

Fig. 13 Running time comparison under the latest distribution

图 13 latest 分布下运行时间对比

4.2.2 读写次数比较

图 14~16 和图 17~19 分别显示了 ZB$^+$-tree 和 CoW B$^+$-tree 的读操作次数和写操作次数对比. ZB$^+$-tree 比 CoW B$^+$-tree 在读操作次数上减少了 25% 左右, 这是因为在删除操作时, 相比于 CoW B$^+$-tree 需要级联的读写, ZB$^+$-tree 在机会主义策略控制的 2 层减少了合并操作并去除了从邻居借键值对的操作. 在写操作次数方面, ZB$^+$-tree 在不同数据规模时平均节约了 75% 的写, 这主要是因为 ZB$^+$-tree 通过 LH 层和 IH 层的独特设计将级联的更新进行了 2 次阻断. 由于 LH 和 IH 都位于 Cov-Zone 中, 允许进行原地更新, 因此节点修改后的地址不发生改变, 所以级联的更新被阻断在了 Cov-Zone 中. 相比于 CoW B$^+$-tree 每次更新至少需要 4 次写操作, ZB$^+$-tree 只需要 1~2 次写操作即可.

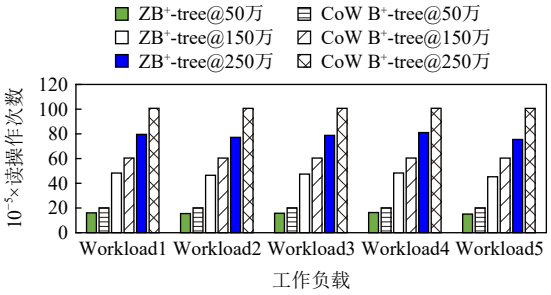

Fig. 14 Read counts comparison under the Zipfian distribution

图 14 Zipfian 分布下读次数对比

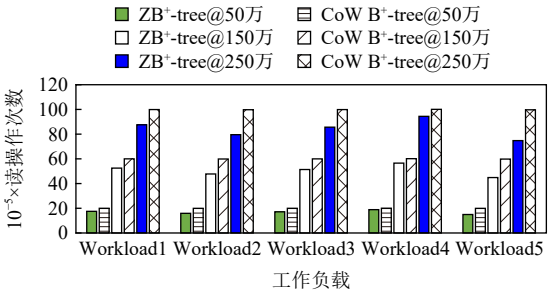

Fig. 15 Read counts comparison under the uniform distribution

图 15 uniform 分布下读次数对比

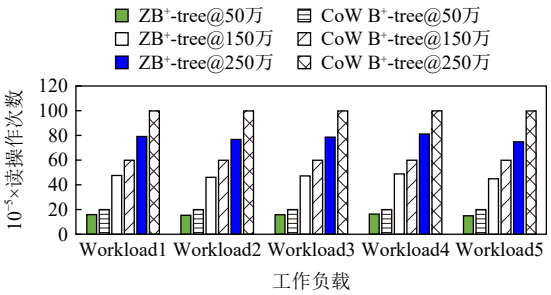

Fig. 16 Read counts comparison under the latest distribution

图 16 latest 分布下读次数对比

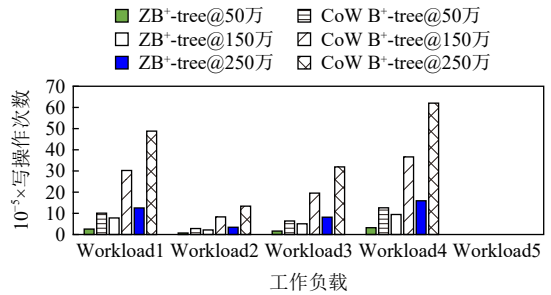

Fig. 17 Write counts comparison under the Zipfian distribution

图 17 Zipfian 分布下写次数对比

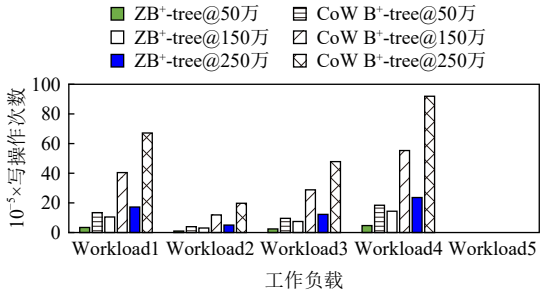

Fig. 18 Write counts comparison under the uniform distribution

图 18 uniform 分布下写次数对比

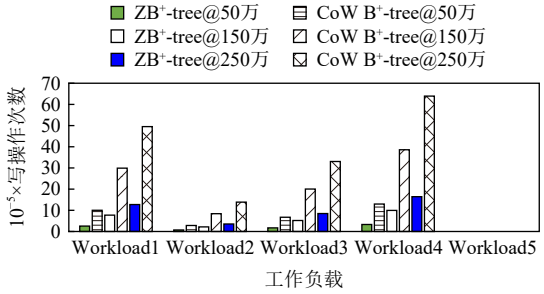

Fig. 19 Write counts comparison under the latest distribution

图 19 latest 分布下写次数对比

4.2.3 空间占用率比较

表 2~6 分别显示了在 Zipfian 分布下，5 种工作负载和 3 种数据规模下 ZB+-tree 和 CoW B+-tree 对于所有 Seq-Zone 的平均占用情况，其他数据分布下实验结果类似. 可以看出，随着数据规模的增加，CoW B+-tree 将快速占用 Seq-Zone 的空间. 在当数据规模为 250 万时，CoW B+-tree 对 Seq-Zone 的占用率最大达到了 0.739 822（Workload4），最小也有 0.450 943（Workload5）. 因此，在大规模写入负载下，CoW B+-tree 的 Seq-Zone 将被快速写满，从而触发耗时的 Zone 垃圾回收和 Zone-Reset 操作，导致系统性能出现急剧下降.

Table 2 Occupancy Rate of Seq-Zone Under Workload1 at Different Data Scales

表 2 不同数据规模下 Workload1 下对 Seq-Zone 的占用率

方案	50 万	150 万	250 万
ZB+-tree	0.000 469	0.001 543	0.002 439
CoW B+-tree	0.120 431	0.402 918	0.678 303

Table 3 Occupancy Rate of Seq-Zone Under Workload2 at Different Data Scales

表 3 不同数据规模下 Workload2 下对 Seq-Zone 的占用率

方案	50 万	150 万	250 万
ZB+-tree	0.000 385	0.001 161	0.001 882
CoW B+-tree	0.086 633	0.300 859	0.513 317

Table 4 Occupancy Rate of Seq-Zone Under Workload3 at Different Data Scales

表 4 不同数据规模下 Workload3 下对 Seq-Zone 的占用率

方案	50 万	150 万	250 万
ZB+-tree	0.000 417	0.001 371	0.002 141
CoW B+-tree	0.103 262	0.353 144	0.599 624

Table 5 Occupancy Rate of Seq-Zone Under Workload4 at Different Data Scales

表 5 不同数据规模下 Workload4 下对 Seq-Zone 的占用率

方案	50 万	150 万	250 万
ZB+-tree	0.000 498	0.001 617	0.002 613
CoW B+-tree	0.132 233	0.432 777	0.739 822

Table 6 Occupancy Rate of Seq-Zone Under Workload5 at Different Data Scales

表 6 不同数据规模下 Workload5 下对 Seq-Zone 的占用率

方案	50 万	150 万	250 万
ZB+-tree	0.000 363	0.001 025	0.001 729
CoW B+-tree	0.073 475	0.262 106	0.450 943

此外，ZB+-tree 对于所有 Seq-Zone 的平均占用率在 5 种不同负载下均低于 0.002 7，远远低于 CoW B+-tree. 而且 ZB+-tree 的 Seq-Zone 占用率不会因为数据规模的增大而出现 Seq-Zone 被快速写满的情况. 因此，在系统运行过程中，ZB+-tree 一般不会频繁触发 Zone 垃圾回收和 Zone-Reset 操作，从而可以在保证系统性能和稳定性的同时节省较多的 Seq-Zone 空间，提升空间效率.

4.2.4 Cov-Zone 和 Seq-Zone 的不同比例

由于当前还没有 ZNS SSD 的真实设备，因此 Cov-Zone 和 Seq-Zone 的比例可能会发生变化，因此，我们改变 Cov-Zone 和 Seq-Zone 的比例来测试 ZB+-tree 的效果.

在本节实验中，我们测试了 3 种不同的 Cov-Zone 和 Seq-Zone 的比例，分别是 1∶32，1∶48，1∶64，数据分布为 Zipfian 分布，数据量设置为 100 万键值对，测试的负载为 Workload1 和 Workload2，如图 20 所示.

从图 20 可以看出，在不同比例下，ZB+-tree 的运行时间都要少于 CoW B+-tree，更改 Cov-Zone 与 Seq-Zone 的比例并不影响读写次数，只会影响对 Seq-Zone 的占用率大小，具体结果如表 7 和表 8 所示.

由表 7 和表 8 可知，在不同比例下，ZB+-tree 对 Seq-Zone 的占用率都远小于 CoW B+-tree. 同时，我们发现更改 Seq-Zone 和 Cov-Zone 的比例对 ZB+-tree 的

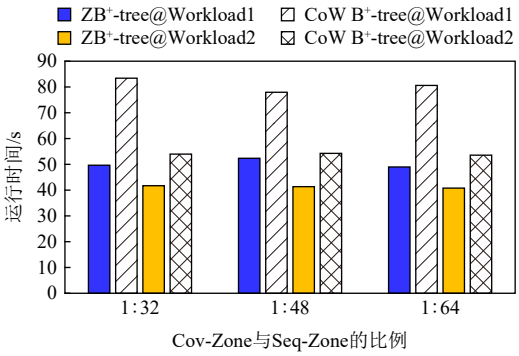

Fig. 20 Comparison of running time under different ratios of Cov-Zone and Seq-Zone

图 20 Cov-Zone 和 Seq-Zone 不同比例下运行时间对比

Table 7 Occupancy Rate of Seq-Zone Under Workload1 at Different Ratios

表 7 不同比例下 Workload1 下对 Seq-Zone 的占用率

方案	1∶32	1∶48	1∶64
ZB+-tree	0.001 125	0.000 751	0.000 562
CoW B+-tree	0.321 145	0.216 282	0.163 043

Table 8 Occupancy Rate of Seq-Zone Under Workload2 at Different Ratios

表 8 不同比例下 Workload2 下对 Seq-Zone 的占用率

方案	1∶32	1∶48	1∶64
ZB+-tree	0.000 959	0.000 638	0.000 478
CoW B+-tree	0.240 339	0.161 861	0.122 018

空间效率影响不大，表明 ZB+-tree 能够适应不同的 Zone 配置.

4.2.5 Zone-Reset 次数比较

在 4.2.1~4.2.4 节的实验中，ZB+-tree 和 CoW B+-tree 都采用了动态选择 Zone 的策略. 该策略将叶子节点和内部节点分开存放以充分利用 ZNS SSD 内部各个 Zone 的空间，并且尽量减少 Zone-Reset 操作，同时将不同特性的节点放在不同的 Zone 来降低访问延迟.

本节的实验旨在进一步验证 ZB+-tree 在不采用动态选择 Zone 策略时的性能. 为了保证实验的公平性，CoW B+-tree 同样不采用动态选择 Zone 的策略. 此外，为了体现实验完整性，本实验中我们通过统计 Zone-Reset 的次数来对比 ZB+-tree 和 CoW B+-tree.

对于 ZB+-tree，设置 1 个 Seq-Zone 和 1 个 Cov-Zone，大小都为 2 GB，然后统计 Zone-Reset 次数. 对于 CoW B+-tree，设置 1 个 Seq-Zone 和 1 个 Cov-Zone（按顺序写模式使用），大小都为 2 GB，当某一个 Zone 写满时，读出有效节点写到另一个 Zone，然后进行 Zone-Reset 操作并统计次数. 实验数据量分别为 250 万、500 万、750 万键值对，数据分布设置为 Zipfian，测试负载为 Workload1 和 Workload2.

实验结果表明，在 Workload1 下，在数据量分别为 250 万、500 万、750 万键值对时，ZB+-tree 均没有触发 Zone-Reset 操作；而 CoW B+-tree 则分别触发了 10, 22, 37 次 Zone-Reset 操作. 在 Workload2 下，ZB+-tree 在数据量为 250 万、500 万、750 万键值对时依然没有触发 Zone-Reset 操作；而 CoW B+-tree 则分别触发了 2, 6, 9 次 Zone-Reset 操作.

因此，CoW B+-tree 比 ZB+-tree 会更容易触发 Zone-Reset 操作，而 ZB+-tree 即使不进行动态选择优化，也能够在插入大量键值时不触发 Zone-Reset 操作. 由于 Zone-Reset 和 Zone 垃圾回收操作的时间延迟高，因此 ZB+-tree 相对于 CoW B+-tree 具有更好的时间性能，尤其是在写密集型负载下，这种优势更加明显.

4.2.6 对 Cov-Zone 的占用分析

在本节实验中，我们创建了 40 个 Seq-Zone 和 1 个 Cov-Zone（每个 Zone 大小都为 2 GB），并分别在 100 万、250 万、500 万、750 万、1 000 万键值对的数据集上运行在 Zipfian 分布下的 Workload1 和 Workload2，然后统计 ZB+-tree 对 Cov-Zone 的空间占用率.

实验结果如图 21 所示. 随着数据规模的增大，ZB+-tree 对 Cov-Zone 的空间占用率呈线性增长趋势，这是因为数据规模越大，负载中涉及更新的键值越多，因此对 Cov-Zone 的使用量也增大. 但是，即使数据规模达到了 1 000 万，对 Cov-Zone 的占用率也没有超过 0.50，这表明 Cov-Zone 的空间不会用满，因此可以满足绝大部分应用的需要.

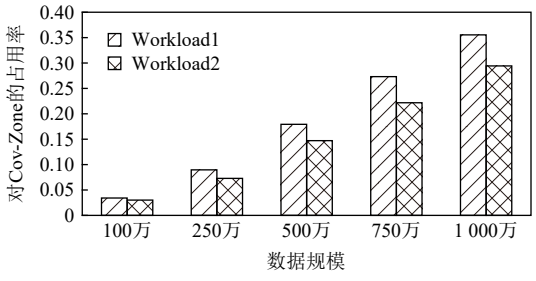

Fig. 21 Changes in the occupancy rate of Cov-Zone by ZB+-tree

图 21 ZB+-tree 对 Cov-Zone 的占用率变化

5 结束语

本文提出了一种 ZNS SSD 感知的新型索引结构 ZB+-tree，通过合理利用 ZNS SSD 中的常规 Zone 和顺序 Zone 的特性来吸收对 Zone 的随机写操作、阻断级联更新、减少读写次数. 我们为 2 种 Zone 内的节

点设计了不同的节点结构，并通过将不同访问频率的节点分散在不同 Zone 中以降低访问延迟，并且满足 ZNS SSD 顺序写的限制. 我们利用 ZNS SSD 模拟器展开了实验，并对比了 ZB$^+$-tree 和传统 CoW B$^+$-tree，结果表明 ZB$^+$-tree 在运行时间和读写次数上均优于 CoW B$^+$-tree. 同时，ZB$^+$-tree 具有更低的 Seq-Zone 空间占用率，可以有效减少系统运行过程中进行的垃圾回收和 Zone-Reset 操作，提高系统性能，并且对 Cov-Zone 的占用率也能够在数据规模增大时始终保持在 0.50 以下，可以满足大多数应用的需要.

未来的工作将主要集中在 3 个方向：1）为 ZB$^+$-tree 增加合理的 GC 机制；2）在选择节点放置时设置更加合理的 Zone 选择方案；3）在真实的 ZNS SSD 设备上进行实验.

作者贡献声明：刘扬提出算法思路，完成实验并撰写论文初稿；金培权提出指导意见并修改论文.

参 考 文 献

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

[2] Han K, Gwak H, Shin D, et al. ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction [C] //Proc of the 15th USENIX Symp on Operating Systems Design and Implementation (OSDI 21). Berkeley, CA: USENIX Association, 2021: 147−162

[3] 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. New York: ACM, 2021: 144−151

[4] 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. New York: ACM, 2021: 21−27

[5] Choi G, Lee K, Oh M, et al. A new LSM-style garbage collection scheme for ZNS SSDs [C] //Proc of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20). Berkeley, CA: USENIX Association, 2020: Article 1

[6] Purandare D R, Wilcox P, Litz H, et al. Append is near: Log-based data management on ZNS SSDs[C/OL] //Proc of the 12th Annual Conf on Innovative Data Systems Research (CIDR'22). 2022[2022-08-20].https://www.ssrc.ucsc.edu/media/pubs/8698b15f3152427d1285a995af615fbe7be26c7b.pdf

[7] Shin H, Oh M, Choi G, et al. Exploring performance characteristics of ZNS SSDs: Observation and implication[C/OL] //Proc of the 9th Non-Volatile Memory Systems and Applications Symp (NVMSA). Piscataway, NJ: IEEE, 2020[2022-08-20].https://ieeexplore.ieee.org/abstract/document/9188086

[8] Park C, Cheon W, Kang J, et al. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications [J/OL]. ACM Transactions on Embedded Computing Systems, 2008 [2022-08-20].https://people.eecs.berkeley.edu/~kubitron/courses/cs262a/handouts/papers/a38-park.pdf

[9] Bux W, Iliadis I. Performance of greedy garbage collection in flash-based solid-state drives[J]. Performance Evaluation, 2010, 67(11): 1172−1186

[10] Smith K. Understanding SSD over-provisioning [EB/OL]. 2022[2022-08-20].https://www.edn.com/understanding-ssd-over-provisioning

[11] Hu Xiaoyu, Eleftheriou E, Haas R, et al. Write amplification analysis in flash-based solid state drives[C/OL] //Proc of 2009 Israeli Experimental Systems Conf (SYSTOR). New York: ACM , 2009[2022-08-20]. http://roman.pletka.ch/publications/hu2009write-ampl.pdf

[12] Bjørling M. ZNS SDDs in Western Digital [EB/OL]. 2020[2022-08-20].https://snia.org/sites/default/files/SDC/2020/075-Bjørling-Zoned-Namespaces-ZNS-SSDs.pdf

[13] Western Digital. NVMe Zoned Namespaces (ZNS) SSDs [EB/OL]. 2021[2022-08-20].https://zonedstorage.io/docs/introduction/zns

[14] NVMe. Zoned Namespaces Command Set [EB/OL]. 2022[2022-08-20].https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-1.1b-2022.01.05-Ratified.pdf

[15] Roh H, Park S, Kim S, et al. B$^+$-tree index optimization by exploiting internal parallelism of flash-based solid state drives[J]. Proceedings of the VLDB Endowment, 2011, 5(4): 286−297

[16] Na G J, Lee S W, Moon B. Dynamic in-page logging for B$^+$-tree index[J]. IEEE Transactions on Knowledge and Data Engineering, 2011, 24(7): 1231−1243

[17] Agrawal D, Ganesan D, Sitaraman R, et al. Lazy-adaptive tree: An optimized index structure for flash devices[J]. Proceedings of the VLDB Endowment, 2009, 2(1): 361−372

[18] Ahn J S, Kang D, Jung D, et al. μ*-tree: An ordered index structure for NAND flash memory with adaptive page layout scheme[J]. IEEE Transactions on Computers, 2012, 62(4): 784−797

[19] Fang Huawei, Yeh M, Suei P, et al. An adaptive endurance-aware B$^+$-tree for flash memory storage systems[J]. IEEE Transactions on Computers, 2013, 63(11): 2661−2673

[20] Jin Peiquan, Yang Chengcheng, Jensen C, et al. Read/write-optimized tree indexing for solid-state drives[J]. The VLDB Journal, 2016, 25(5): 695−717

[21] Lv Yanfei, Li Jing, Cui Bin, et al. Log-compact R-tree: An efficient spatial index for SSD[C]//Proc of the Int Conf on Database Systems for Advanced Applications (DASFAA 11). Berlin: Springer, 2011: 202−213

[22] Jin Peiquan, Yang Chengcheng, Wang Xiaoliang, et al. SAL-Hashing: A self-adaptive linear hashing index for SSDs[J]. IEEE Transactions on Knowledge and Data Engineering, 2020, 32(3): 519−532

[23] Ho V, Park D. WPCB-tree: A novel flash-aware B-tree index using a

write pattern converter [J/OL]. Symmetry, 2018[2022-08-20].https://www.mdpi.com/2073－8994/10/1/18/pdf

[24] Jin R, Cho H, Lee S, et al. Lazy-split B^+-tree: A novel B^+-tree index scheme for flash-based database systems[J]. Design Automation for Embedded Systems, 2013, 17(1): 167−191

[25] Twigg A, Byde A, Miłoś G, et al. Stratified B-trees and versioned dictionaries[C] //Proc of the 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11). Berkeley, CA: USENIX Association, 2011: Article 3

[26] Yan Wei, Zhang Xingjun, Ji Zeyu, et al. One-direction shift B^+-tree based on persistent memory[J]. Journal of Computer Research and Development, 2021, 58(2): 371−383 (in Chinese)
(闫玮, 张兴军, 纪泽宇, 等. 基于持久性内存的单向移动B^+树[J]. 计算机研究与发展, 2021, 58(2): 371−383)

[27] Western Digital. null_blk [EB/OL]. 2022[2022-10-15].https://zonedstorage.io/docs/getting-started/zns-device

[28] Western Digital. libzbd [EB/OL]. 2022[2022-08-20].https://github.com/westerndigitalcorporation/libzbd

[29] 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

Liu Yang, born in 2000. Master candidate. His main research interest includes database technology for new hardware.

刘　扬, 2000 年生. 硕士研究生. 主要研究方向为面向新型硬件的数据库技术.

Jin Peiquan, born in 1975. PhD, associate professor, PhD supervisor. Distinguished member of CCF. His main research interests include databases and big data.

金培权, 1975 年生. 博士, 副教授, 博士生导师. CCF 杰出会员. 主要研究方向为数据库与大数据.